НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

«На правах рукопису»
УДК ____004.8_____

До захисту допущено:

Завідувач кафедри

_____ Олександр, РОЛІК

«___»_____2021р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Інженерія програмного забезпечення комп'ютерних систем»

зі спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Штучний інтелект на основі нейронної мережі для гри в жанрі стратегія в реальному часі»

Виконала:
студент VI курсу, групи ІТ-91мн
Свириденко Олександр Андрійович _____
Керівник:
Доцент кафедри АУТС, к.т.н., доцент
Дорогий Ярослав Юрійович _____
Рецензент:
доцент кафедри ПЗКС, к.т.н., доцент
Василь ЦУРКАН _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2021 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-наукова програма «Інженерія програмного забезпечення комп'ютерних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Олександр, РОЛІК

«___»_____2021 р.

ЗАВДАННЯ

на магістерську дисертацію студенту

Свириденко Олександра Андрійовича

1. Тема дисертації «Штучний інтелект на основі нейронної мережі для гри в жанрі стратегія в реальному часі», науковий керівник дисертації Дорогий Ярослав Юрійович, доцент кафедри АУТС, к.т.н., затверджені наказом по університету від «12» березня 2021 р. № 809-с
2. Термін подання студентом дисертації 11.05.2021
3. Об'єкт дослідження штучний інтелект в іграх жанру стратегій в реальному часі
4. Предмет дослідження побудований з використанням штучних нейронних мереж штучний інтелект для гри в жанрі стратегії в реальному часі, , зокрема модулі мікроуправління (тактичний) та макроуправління (стратегічний), з використанням штучних нейронних мереж в комбінації з іншими підходами, а також розподілена система навчання штучного інтелекту з використанням централізованого серверу
5. Перелік завдань, які потрібно розробити: проаналізувати теоретичні матеріали, визначити вимоги та задачі до штучного інтелекту, спроектувати модулі штучного інтелекту для мікро- та макро-управління, спроектувати та реалізувати тестове середовище, перевірити ефективність штучного інтелекту в рамках розробленого середовища
6. Орієнтовний перелік графічного (ілюстративного) матеріалу: діаграма класів, діаграма компонентів, діаграма структури бази даних, структурна діаграма, функціональна діаграма, діаграма послідовності для початкового етапу, структура класів для нейронної мережі,

7. Орієнтовний перелік публікацій Тактичний штучний інтелект з використанням нейронної мережі для стратегії в реальному часі; Перспективи розвитку штучного інтелекту для ігор в жанрі стратегій в реальному часі;

9. Дата видачі завдання 01.02.2021

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Вибір та узгодження теми магістерської дисертації	02.02.2021-04.02.2021	
2	Аналіз теоретичних матеріалів та вивчення предметної області	05.02.2021-18.02.2021	
3	Розробка технічного завдання	15.03.2021-18.03.2021	
4	Аналіз існуючих рішень та підходів до розробки штучного інтелекту	19.03.2021-25.03.2021	
5	Вибір технологій реалізації штучного інтелекту для мікро- та макроуправління	26.03.2021-28.03.2021	
6	Розробка структури тестового середовища та серверної частини	01.04.2021-04.04.2021	
7	Реалізація, перевірка та налагодження програми	05.04.2021-23.04.2021	
8	Оформлення текстових та графічних матеріалів	15.04.2021-04.05.2021	
9	Передзахист магістерської дисертації	05.05.2021	
10	Доопрацювання пояснювальної записки та підготовка презентації	06.05.2021-11.05.2021	
11	Подача магістерської дисертації	11.05.2021	

Студент Олександр, СВИРИДЕНКО

Науковий керівник Ярослав, ДОРОГИЙ

РЕФЕРАТ

Магістерська дисертація присвячена розробці та опису штучного інтелекту з використанням нейронних мереж для гри жанрі стратегія в реальному часі.

Магістерська дисертація міститься на 118 сторінках та включає 44 рисунки, 5 таблицю та 31 бібліографічні посилання. Вона складається з наступних розділів: вступ, 5 розділів для основної частини, висновки, перелік посилань та 8 додатків.

Актуальність обраної теми полягає в підвищенні ефективності агентів штучного інтелекту для ігор в жанрі стратегій в реальному часі, а також використання розподіленої системи з централізованим сервером як елементу покращення ефективності модулів штучного інтелекту.

Метою роботи є створення системи агентів штучного інтелекту з використанням штучних мереж для забезпечення високої ефективності роботи ботів в іграх жанру стратегій в реальному часі.

Об’єктом дослідження є штучний інтелект в іграх жанру стратегій в реальному часі, зокрема модулі мікроуправління (тактичний) та макроуправління (стратегічний), з використанням штучних нейронних мереж в комбінації з іншими підходами.

Предметом дослідження є побудований з використанням штучних нейронних мереж штучний інтелект для гри в жанрі стратегії в реальному часі, а також розподілена система навчання штучного інтелекту з використанням централізованого серверу.

Ключові слова: нейронні мережі, штучний інтелект, ієрархічна мережа задач, стратегії в реальному часі, розподілена система навчання нейронних мереж.

SUMMARY

The master's dissertation is devoted to the development and description of artificial intelligence using neural networks for the game genre of real-time strategy.

The master's dissertation is on 118 pages and includes 44 figures, 5 tables and 31 bibliographic references. It consists of the following sections: introduction, 5 sections for the main part, conclusions, list of references and 8 appendices.

The relevance of the chosen topic is to increase the efficiency of artificial intelligence agents for games in the genre of real-time strategy, as well as the use of a distributed system with a centralized server as an element of improving the efficiency of artificial intelligence modules.

The aim of the work is to create a system of artificial intelligence agents using artificial networks to ensure high efficiency of bots in games genre strategy in real time.

The object of research is artificial intelligence in real-time strategy games, including modules of microcontrol (tactical) and macrocontrol (strategic), using artificial neural networks in combination with other approaches.

The subject of the study is artificial intelligence built using artificial neural networks to play in the genre of real-time strategy, as well as a distributed system of artificial intelligence training using a centralized server.

Keywords: neural networks, artificial intelligence, hierarchical network of tasks, real-time strategies, distributed neural network learning system.

ЗМІСТ

ВСТУП

Зараз ми можемо спостерігати надзвичайно активний розвиток ігрової індустрії. З кожним днем з'являються нові проекти, прибутки компаній ростуть, а інтерес у користувачів стає дедалі вищим. Нові проекти, в яких реалізовано сміливі ідеї і незвичайні ідеї, знаходять своїх користувачів, а інколи навіть і стають засновниками цілих жанрів ігор. Деякі жанри, що донедавна здавалися неприбутковими і забутими, повертаються до користувачів у продуктах, створених невеликими компаніями. Це можна сказати і про стратегії в реальному часі, досить специфічний жанр, який поєднує в собі складність стратегічного мислення, тактичного управління об'єктами та швидкої реакції на події, що відбуваються на ігровому полі.

Для стратегій в реальному часі проблема штучного інтелекту особливо актуальна, оскільки складність управління і прийняття рішень призводить до того, що розробники закладають в штучний інтелект певний послідовний алгоритм дій, який використовується раз за разом. Звичайно, для гравців таке протистояння не є надто цікавим, адже гравець постійно змагається з одним і тим же алгоритмом, з його помилками та особливостями, вивчивши які перемога стає тільки питанням часу та бажання. Окремо можна виділити питання складності штучного інтелекту та швидкості його дій, схема з легким, середнім та складним рівнем не відповідає реальним потребам гравців.

Актуальність обраної теми полягає в підвищенні ефективності та адаптивності агентів штучного інтелекту для ігор в жанрі стратегій в реальному часі, а також використання розподіленої системи з централізованим сервером як елементу покращення ефективності модулів штучного інтелекту.

Метою роботи є створення системи агентів штучного інтелекту з використанням штучних мереж для забезпечення високої ефективності роботи ботів в іграх жанру стратегій в реальному часі.

Об'єктом дослідження є штучний інтелект в іграх жанру стратегій в реальному часі.

Предметом дослідження є побудований з використанням штучних нейронних мереж штучний інтелект для гри в жанрі стратегії в реальному часі, зокрема модулі мікроуправління (тактичний) та макроуправління (стратегічний), з використанням штучних нейронних мереж в комбінації з іншими підходами, а також розподілена система навчання штучного інтелекту з використанням централізованого серверу.

Наукова новизна полягає у вирішенні науково-практичного завдання розроблення ефективного штучного інтелекту для стратегій в реальному часі. Результати роботи можуть бути використані у подальших дослідженнях штучного інтелекту для ігор інших жанрів, а також можуть бути впроваджені в нові або вже існуючі ігрові продукти як ефективніша альтернатива існуючому штучному інтелекту.

Для тестування бота необхідне середовище з певними ігровими правилами, тому для визначення реальної ефективності штучного інтелекту необхідно порівняти його з іншими ботами. Окрім тестового середовища важливим елементом сучасних систем штучного інтелекту є сервер, що керує і навчає моделі. Кожен з цих компонентів потрібно спроектувати та розробити для перевірки ефективності штучного інтелекту.

За результатами роботи були опубліковані тези до конференції «XIX International Scientific and Practical Conference MODERN SCIENCE» 11-12 травня, м. Хьюстон, США, на тему «Перспективи розвитку штучного інтелекту для ігор в жанрі стратегій в реальному часі», та конференції «LXVI International Scientific and Practical Conference INNOVATIONS OF THE SCIENCE XX CENTURY» 17 травня, м. Дніпро, Україна, на тему «Тактичний штучний інтелект з використанням нейронної мережі для стратегії в реальному часі».

1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ДЛЯ ШТУЧНОГО ІНТЕЛЕКТУ

Штучний інтелект активно використовується в різних сферах, від медицини до економіки. Однак не дивлячись на таке активне поширення, найперспективнішою сферою для розвитку та аналізу результативності штучного інтелекту залишаються комп'ютерні ігри. Особливо серед різних жанрів варто виокремити стратегії в реальному часі. Причиною цього є складність задач, які ігри в даному жанрі ставлять перед гравцем та штучним інтелектом.

Штучний інтелект для гри в шахи, шашки або інші настільні ігри і його можливості до гри на рівні людини для нас є звичним. До того ж, після перемоги штучного інтелекту DeepBlue над гросмейстером Каспаровим [1], дослідники довели, що штучний інтелект в покрокових іграх може перемогти навіть професійних гравців найвищого класу. З іграми в жанрі стратегій в реальному часі ситуація зовсім інша. На відміну від покрокових ігор, де гравці ходять по черзі і бачать зміни позицій супротивника, в стратегіях в реальному часі ситуація міняється кожної секунди, гравці діють синхронно, а одним з ключових елементів є «туман війни» - відсутність видимості в частинах карти, де у гравця немає підконтрольних підрозділів або будівель. Через це значно ускладняється можливість аналізу поточної ситуації, відповідно зростає складність визначення подальших дій. Один з дослідників штучного інтелекту для стратегій в реальному часі, Майкл Буро, сказав: «Щоб зрозуміти всю складність стратегій в реальному часі, уявіть собі гру в шахи на дошці розмірами 512 на 512 і сотнями фігур, які синхронно і повільно рухаються». [2]

1.1 Напрямки використання штучного інтелекту в жанрі RTS

Популярні комерційні проекти в жанрі стратегій в реальному часі мають типові правила, що поширюються на весь жанр. Зазвичай на початку гри у

кожного є центр і декілька небойових юнітів – працівників – які будують будівлі і добувають ресурси. В залежності від потреб будівлі бувають різних типів: ресурсні, технологічні, захисні або виробники юнітів. В залежності від обраної стратегії гравець може визначити, яка будівля або юніт потрібні на даний момент. Прийняття цих рішень має великий вплив на подальшу гру і залежать від стратегії гравця в цілому. За цю частину, яку також дослідники називають *макроконтроль*, у бота відповідає *модуль прийняття стратегічних рішень*.

Після закінчення будівництва споруд, в яких можна проводити найняття бойових одиниць, гравець отримує доступ до армії. Армія складається з юнітів різного типу та призначення. Юнітів потрібно використовувати відповідно до їх характеристик – одиниці, призначені для стримування ворога потрібні на передовій, одиниці підтримки мають триматись в тилу. За управління військами під час сутичок, або *мікроконтроль*, відповідає *модуль прийняття тактичних рішень*.

З самого початку гравцю доступна тільки невелика частина карти – більшість потрібно розвідати. Також гравцю невідома позиція або позиції супротивників та поточна ситуація з будівлями та військами. Для цього потрібна розвідка – направлення своїх юнітів в частини карти, що закриті «туманом війни» та аналіз ситуації. За це відповідає *модуль розвідки та визначення планів*.

Окремо варто зазначити, що окрім вищезгаданих трьох модулів штучного інтелекту в стратегіях в реальному часі, ще існує навчання ботів для коректного реагування на ситуації. Цю частину штучного інтелекту дослідники найчастіше поєднують з модулем визначення планів супротивника.

1.2 Прийняття тактичних рішень

Для створення тактичного штучного інтелекту у бота дослідники та інженери використовують різні підходи, найчастішими варіантами є:

- Навчання з підкріпленням;
- Пошук по ігровому дереву;
- Баєсова модель;
- Рішення на основі прецедентів;
- Нейронні мережі.

Навчання з підкріпленням це галузь машинного навчання, де агент повинен обирати дії, які приведуть до максимізації умовної винагороди. На відміну від навчання з учителем, агенту не задаються правильні варіанти вхідних та вихідних даних, а неоптимальні рішення агента не виправляються.

Для гри StarCraft в 2011 році [3] дослідниками було створено модуль для прийняття тактичних рішень, в основі якого була комбінація нейронної мережі та алгоритму навчання з підкріпленням, за допомогою якої реалізовувалось тактичне управління кожним з солдатів. Для малих груп алгоритм показав себе як набагато ефективніший в порівнянні з вбудованими сценаріями (рис. 1.1).

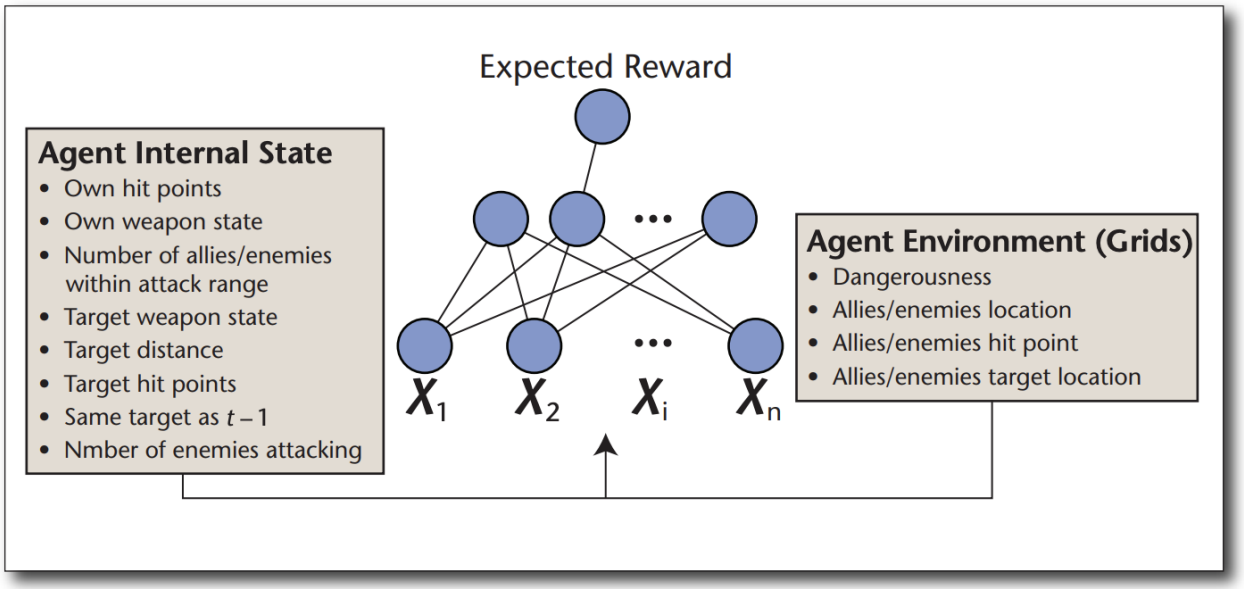

Рисунок 1.1 – Мережа для визначення потенційної очікуваної нагороди для окремої тактичної одиниці.

В 2017 році дослідником було розглянуто варіанти використання класичного навчання з підкріпленням для стратегій в реальному часі, а також запропоновано вирішення деяких проблем, наприклад, вирішення проблема з фактором великого розгалуження [4].

Інша команда вчених з Нідерландів в 2018 році використала навчання з підкріпленням для тестування у власному середовищі [5]. В агенті було поєднано ієрархічне навчання з підкріпленням з багаторівневим перцептроном, який оброблював високо рівневі команди для покращення якості навчання системи.

Рішення на основі прецедентів часто використовуються в стратегічних іграх, де мікроконтроль важливіший за стратегічне планування. Прийняття рішень на основі прецедентів в широкому сенсі є вибором дії на основі вже відомих послідовностей. Створений у 2009 році дослідниками модуль для гри Warcraft [6] зміг ефективно показати себе у мікроконтролі в битвах з вбудованим штучним інтелектом та стати ефективним помічником гравцю, забезпечуючи ефективне мікроуправління військами, тим самим надаючи

гравцю можливість сконцентруватись на високо-рівневому плануванні – стратегічному. Рішення на основі прецедентів також можна поєднати з навчанням з підкріпленням [7] для ефективного використання юнітів і їх взаємодії на тактичному рівні.

Пошук по ігровому дереву є одним з алгоритмів, які нечасто використовують для стратегічних рішень, однак він може гарно себе показати для прийняття тактичних рішень.

Дослідниками [8] було створено систему, яка, аналізуючи всім відомі стратегії, шукала рівновагу Неша – за умов цієї рівноваги гравець не може отримати переваг (збільшити виграш) змінивши стратегію в односторонньому порядку. Для дослідження було обрано лише тактичну складову, а також виключено мікроменеджмент кожного окремого юніта – управління відбувалося групами і концентрувалось на переміщенні. Як результат система змогла перемогти бота у грі практично у всіх ситуаціях, адже на кожну конкретно визначену розробниками стратегію ігрового бота система відповідала покращеною стратегією.

Для ігор закритим вихідним кодом (а це більшість комерційних проектів) досить важко використовувати алгоритми, що базуються на пошуку. Тому у 2012 році дослідники для тестування штучного інтелекту на основі пошуку по ігровому дереву в грі StarCraft створили аналог гри, названий SparCraft, з дещо спрощеними правилами, але загалом таким самим ігровим процесом [9]. Результати тестів показали ефективність агента в 92% випадків, враховуючи обмеженість часу на вибір рішення – всі обчислення проходили в реальному часі

Баєсова модель, яка представляє собою набір випадкових змінних та залежностей між ними за допомогою орієнтованого ациклічного графу, може бути використана також в тактичному прийнятті рішень, для визначення напрямку руху юнітів [10].

Нейроеволюція – це техніка використання еволюційних алгоритмів для навчання нейронної мережі. Підхід rtNEAT [11] може бути використаний для

впливу як на коефіцієнти, так і на топологію моделі нейронної мережі для контролю кожного окремого юніта. На вхід до моделі потрапляють дані, які юніт отримує з навколишнього середовищі, а також від сусідніх юнітів. Кожен носій нейромережі мав визначену функцію поточної ефективності, яка залежала від позиції, здоров'я, рівня та інших параметрів. Юніти, які показували слабкі дані ефективності в бою замінялись більш ефективними одиницями. Тестування проводилось в досить зручних для штучного інтелекту умовах, однак він показав свою ефективність при однаковій кількості юнітів і однаковому типі з обох сторін.

Загалом використання нейронних мереж в побудові агентів для прийняття тактичних рішень є поширеною практикою. Наприклад, дослідники за допомогою згорткової нейронної мережі (convolutional neural network, CNN) змогли досягти значно кращих результатів у порівнянні з іншими алгоритмами[12]. До того ж варто зауважити що такі нейронні мережі працюють повільніше ніж більшість алгоритмів, які активно тестуються в наш час, але не дивлячись на це згорткові нейронні мережі показують кращий результат. Окрім того, дослідники припускають, що в поєднанні останніх досягненні в розвитку ієрархічного пошуку та згорткових нейронних мереж є ключем до перемоги над людьми-гравцями в такому складному для агентів штучного інтелекту просторі стратегій в реальному часі.

Згорткові нейронні мережі також використовуються для таких завдань як визначення переможця в поточній ситуації [13]. Для тестування рішення було використано μRTS (мікро стратегія в реальному часі), а для нейронної мережі було використано специфічний вид – багатомірні згорткові нейромережі.

1.3 Прийняття стратегічних рішень

Стратегічні рішення, або макроменеджмент, є високорівневою частиною штучного інтелект в стратегічних іграх і впливають на гру в

довгостроковій перспективі – досліджена технологія або побудована будівля на початку гри може змінити ситуацію і вплинути на розвиток подій в самому кінці гри. Для визначення стратегічних дій використовують планувальні системи – вони показали свою ефективність в реальних продуктах і наукових дослідженнях. Складність проблеми полягає у залежності стратегічних рішень від рішень супротивника. Однак «туман війни» не дозволяє знати, що є на базі у супротивника та який розмір армії. Через такі проблеми рішення доводиться приймати керуючись неповними або навіть відсутніми даними, що може суттєво вплинути на наслідки. Основними підходами є планування на основі прецедентів, ієрархічне планування та автономне досягнення цілей. Рідше використовуються дерева поведінки або еволюційні системи.

Базовими варіантами штучного інтелекту для ігор є створення набору цілей з умовною винагородою. Такі підходи використовувались в перших іграх в жанрі стратегій в реальному часі і завданням бота було набрати як можна більшу кількість балів виконуючи різні задачі і досягаючи різних цілей. Цінність задачі змінювалась в ході гри залежно від поточної ситуації.

Прийняття рішень на основі прецедентів в стратегічному модулі має подібну схему до тактичного: розробники створюють певний набір відомих їм послідовностей дій в залежності від поточної ситуації, а штучний інтелект шукає найбільш подібні варіанти для поточної ситуації і використовує їх. Перевагами такого підходу в порівнянні з іншими є його адаптивність до нових стратегій і відсутність потреби у постійному навчанні.

Вперше штучний інтелект для прийняття стратегічних рішень з використанням прецедентів був створений дослідниками у 2005 [14]. Важливою перевагою була адаптивність до різних стратегій супротивника – в той час більшість агентів штучного інтелекту показували хороші результати тільки проти «статичного» супротивника, який мав одну заскриптовану стратегію. При зміні стратегії супротивника агент мав проходити додатковий етап навчання і адаптації.

Поєднання прийняття рішень на основі прецедентів з ідеєю нечітких множин дозволило абстрагувати інформацію про стан [15]. Це дозволило дослідникам значно спростити простір станів. Вибір стратегічних рішень керується кількістю будівель та одиниць.

Ієрархічне планування – це система розподілення цілей в грі в ієрархічну мережу задач, де присутні як основні високо рівневі задачі (перемога над супротивником), так і задачі низького рівня, які є підзадачами для високорівневих (найняти робочого). Ієрархічне планування часто поєднують з іншими алгоритмами для стратегічного планування, наприклад з рішеннями на основі прецедентів.

Часто ієрархічне планування дослідники формалізують у побудову чітко визначеної і структурованої ієрархічної мережі задач. В такій мережі високорівненві задачі можуть бути розділені на ряд послідовних дій, що будуть виконані агентом [16].

Ієрархічна мережа задач була використана для створення агента прийняття стратегічних рішень в стратегії з відкритим вихідним кодом Spring [17]. Така архітектура агента дозволяла йому ефективно реагувати на втрату будівель або краще добувати ресурси. Підхід показав високу ефективність проти вбудованого штучного інтелекту.

Іншим підходом з використанням ієрархічного планування є створення мережі послідовних, паралельних та умовних дій, які виконуються в залежності від поточного стану [18]. Всі рішення синхронізуються через умовну «дошку», де зберігаються результати виконання попередніх задач.

Автономне досягнення цілей є технікою вибору послідовностей дій, які змінюються в залежності від ситуації. При отриманні нових даних агент може змінити послідовність дій, які позначатимуть виконання цілі [19]. Це дозволяє боту ефективно відповідати на різкі зміни у грі та бути гнучким у досягненні цілей, при цьому не переключаючись з них. Цей підхід часто комбінують з іншими, наприклад з прийняттям рішень на основі прецедентів.

Для побудови агента дослідники створили систему з використанням ABL (A Behavior Language). Особливістю системи була можливість коригування планів у разі виникнення несподіваних подій. Також плани могли виконуватись паралельно, якщо вони були незалежними [20].

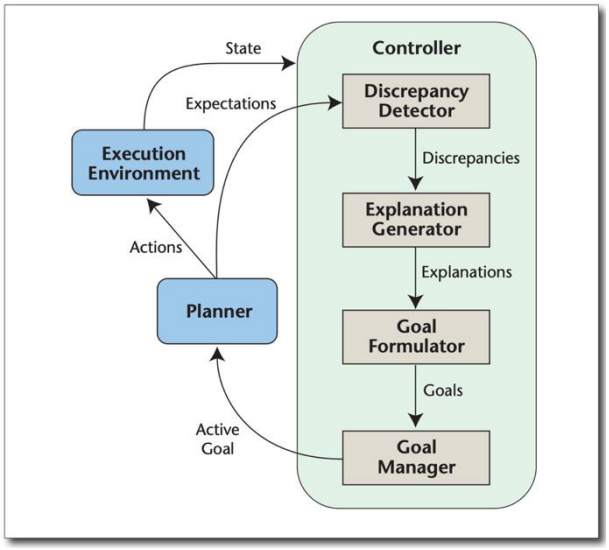

Рисунок 1.2 – Схема автономного прийняття рішень.

1.4 Розвідування і визначення планів

Зі стратегічного планування як окремий модуль можна виокремити техніку визначення плану супротивника та його стратегії. Через технологію «туману війни», яку часто використовують в стратегіях, визначення планів майже завжди базується на неповній інформації, тому часто техніки використовують уже готову базу знань або навчальний модуль. В цій частині будуть розглянуті такі техніки:

- Дедуктивні;
- Абдуктивні;
- Ймовірнісні;
- Прецедентні.

Визначення плану дедуктивним методом визначає план за допомогою співставлення поточної ситуації з гіпотетичною ситуацією з набору відомих

планів. За допомогою дедуктивного методу можна визначити план супротивника навіть з обмеженої кількості інформації, оскільки висновки можна зробити навіть з перших починань підготовки певного плану[21].

Розроблене дослідниками [22] дерево рішень для гри Starcraft також можна вважати однією з реалізацій дедуктивного методу для розпізнавання планів. Для тренування системи було використано базу повторів ігор, з якої основну увагу було приділено послідовності створення будівель та технологій.

Абдуктивний метод полягає у можливості вивести з комбінації гіпотези та висновку додаткові гіпотези. На базі цих даних планувальник створює ціль, яка буде скоригована якщо гіпотези не підтверджено або нові дані про супротивника вказують на необхідність використання інших дій та постановки нових або розширення списку поточних цілей. Дослідниками було створено стратегічний штучний інтелект з використанням технології автономного досягнення цілей та абдуктивним методом для визначення плану та стратегії супротивника[20]. Такий метод потребує постійного покращення бази цілей та нових умов для постановки точних цілей або кращого коригування існуючих.

Ймовірнісні методи використовують статистику і ймовірнісні оцінки дій та їх результатів у різні проміжки часу. Для побудови моделі ймовірнісної стратегії розвитку супротивника через аналіз порядку побудови будівель використовують записи ігор реальних гравців без попереднього навчання.

Ймовірнісні методи використовуються з моделями Маркова – розвиток гравця представлено набором станів і зв’язками (потенційними переходами) між цими станами [23]. Варто зауважити, що використаний підхід залежить від специфіки фракції (раси/нації) і використовуються на початкових етапах гри.

Попереднє дослідження отримало свій розвиток з додаванням динамічної баєсової моделі та враховувало розвідку ворожої бази та потенційно не розвідані ділянки [24].

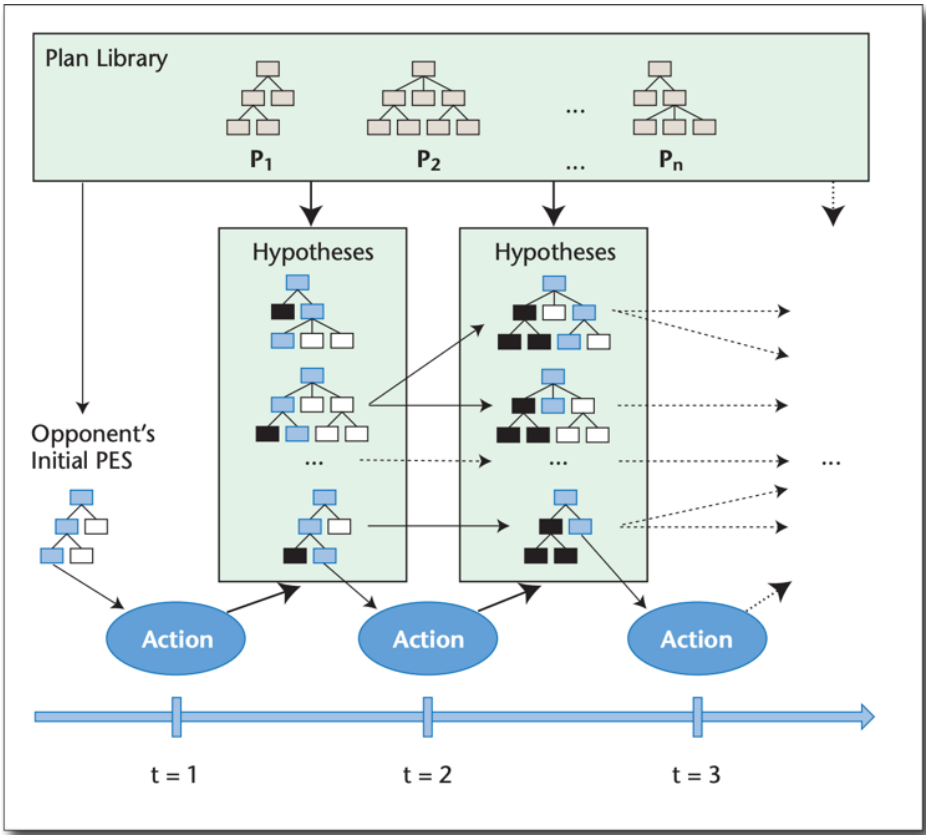

Рисунок 1.3 – Візуалізація абдуктивних методів для визначення плану

Прецедентні методи дозволяють отримувати дані про можливі дії супротивників через розгляд їх стратегії в базі минулих випадків та потенційних можливостей. Такий варіант особливо ефективно можна використати на початку гри, адже початковий етап яскраво вказує на майбутню стратегію гравця, оскільки початкове визначення позицій є основним у грі. Наприклад, для таких ігор як StarCraft, створені схеми початкового розвитку, вони вказують послідовність дій для досягнення результату в певній сфері – захоплення додаткових ресурсних баз, створення великої кількості бойових одиниць, розвиток технологічної складової, тощо. Проаналізувавши навіть неповні дані на початковому етапі гри гравець або бот може досить чітко визначити які дії були виконані супротивником і яка стратегія найбільш вірогідна.

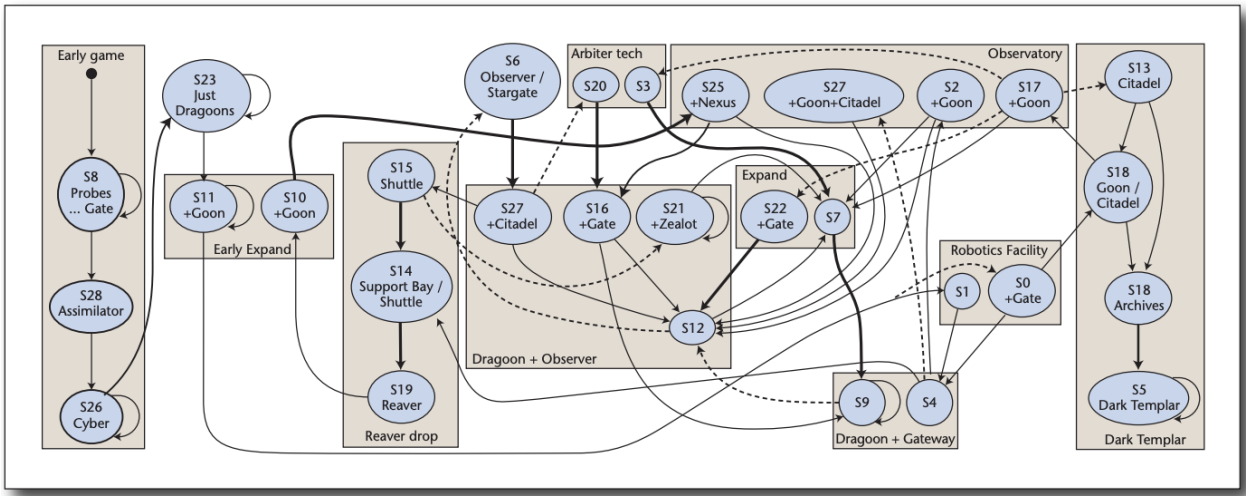

Рисунок 1.4 – Один з варіантів початкового набору дій для раси Protos в грі Starcraft

1.5 Висновки до розділу

Штучний інтелект для ігор в жанрі стратегій в реальному часу складається з трьох основних модулів: тактичного управління, стратегічного управління та розвідки і визначення планів.

Через специфіку задач, які модуль штучного інтелекту повинен вирішувати, алгоритми та підходи для створення кожного мають досить серйозні відмінності, що не дозволяють використовувати один алгоритм для прийняття всіх рішень.

Серед існуючих алгоритмів для створення тактичного штучного інтелекту дослідники виділяють нейронні мережі, стратегічного – комбінацію рішень на основі прецедентів та ієрархічного планування.

2 ПРОЕКТУВАННЯ ШТУЧНОГО ІНТЕЛЕКТУ

2.1 Вибір алгоритмів та підходів

Проаналізувавши велику кількість досліджень та матеріалів щодо підходів для створення штучного інтелекту для ігор в жанрі стратегія в реальному часі, було зроблено висновок, що найбільш гнучкі і якісні рішення використовують комбінацію декількох варіантів (підходів) до створення різних частин штучного інтелекту.

Для так званого тактичного штучного інтелекту найбільш перспективною виглядає технологія нейронних мереж, що дозволить йому реагувати правильно навіть у цілком незнайомій ситуації.

Розглянемо специфіку побудови класичної нейронної мережі, що має назву багаторівневий перцептрон [26].

Штучні нейронні мережі (ШНМ) - це програмна імплементація нейронних структур нашого мозку. Вони можуть змінювати тип переданих сигналів в залежності від електричних або хімічних сигналів, які в них передаються. Нейронна мережа у людському мозку - величезна взаємопов'язана система нейронів, де сигнал, який передається одним нейроном, може передаватися у тисячі інших нейронів. Навчання відбувається через повторну активацію деяких нейронних з'єднань. Через це збільшується імовірність виведення потрібного результату при відповідній вхідній інформації (сигналах). Такий вид навчання використовує зворотний зв'язок - при правильному результаті нейронні зв'язки, які виводять його, стають більш щільними.

Штучні нейронні мережі імітують поведінку мозку у простішому вигляді. Вони можуть бути навчені контрольованим та неконтрольованим шляхами. У контрольованій ШНМ, мережа навчається шляхом передавання відповідної вхідної інформації та прикладів вихідної інформації. Наприклад, спам-фільтр у електронній поштовій скриньці: вхідною інформацією може

бути список слів, які зазвичай містяться у спам-повідомленнях, а вихідною інформацією - класифікація для відповідного повідомлення (спам, чи не спам). Такий вид навчання додає ваги зв'язкам ШНМ, але це буде обговорено пізніше.

Біологічний нейрон імітується у ШНМ через активаційну функцію. У задачах класифікації (наприклад визначення спам-повідомлень) активаційна функція повинна мати характеристику "вмикача". Іншими словами, якщо вхід більше, ніж деяке значення, то вихід повинен змінювати стан, наприклад з 0 на 1 або -1 на 1. Це імітує "включення" біологічного нейрону. У якості активаційної функції зазвичай використовують сигмоїдну функцію:

$$f(z) = \frac{1}{1+\exp(-z)} \qquad (2.1)$$

Графік даної функції виглядає таким чином (рис. 2.1).

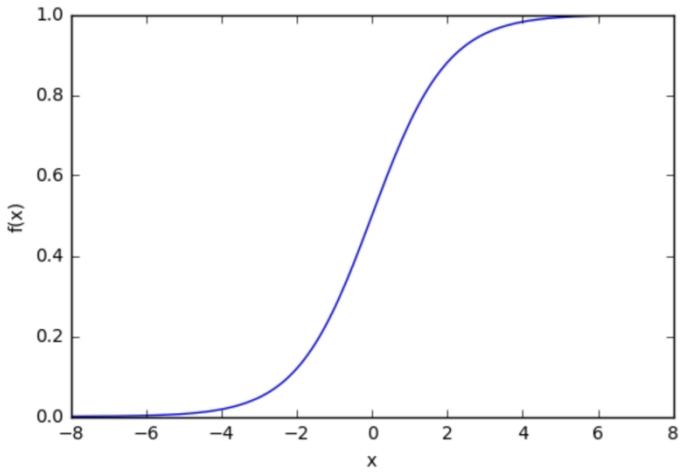

Рисунок 2.1 – Сигмоїдна функція (графік)

З графіку можна побачити, що функція "активаційна" – вона росте з 0 до 1 з кожним збільшенням значення x. Сигмоїдна функція є гладкою і неперервною. Це означає, що функція має похідну, що у свою чергу є дуже важливим фактором для навчання алгоритму.

Як було згадано раніше, біологічні нейрони ієрархічно з'єднані в мережамі, де вихід одних нейронів є входом для інших нейронів. Ми можемо представити такі мережі у вигляді з'єднаних шарів з вузлами. Кожен вузол приймає зважений вхід, активує активаційну функцію для суми входів, та генерує вихід.

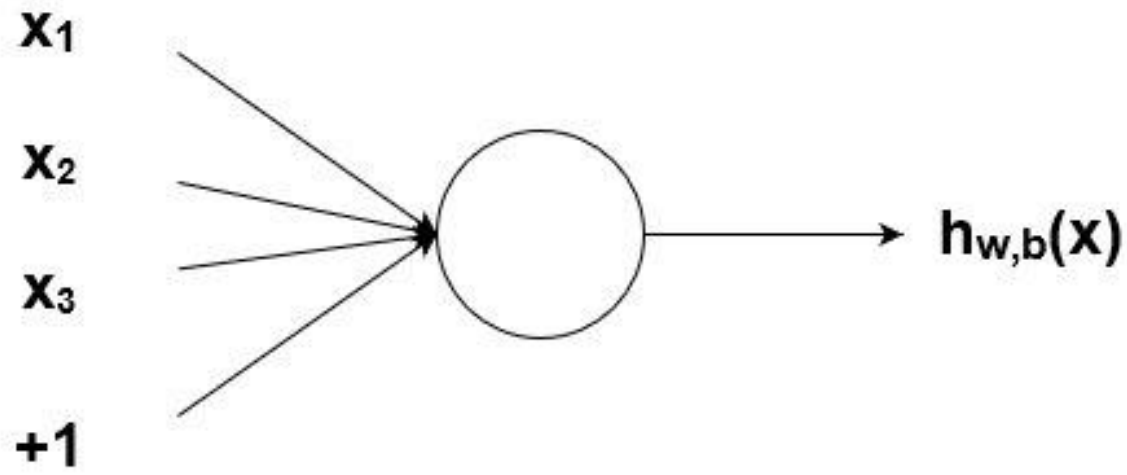

Рисунок 2.2 – Штучний нейрон з входами попереднього вузла x1…x3 та зміщенням 1

Коло на картинці зображує вузол. Вузол є "місцерозташуванням" активаційної функції, він приймає зважені входи, сумує їх, а потім вводить їх в активаційну функцію. Вивід активаційної функції представлений через h. Примітка: у деякій літературі вузол також називають персептроном. За вагу беруться числа (не бінарні), які потім множаться на вході і сумуються у вузлі. Зважений вхід у вузол має вигляд:

$$x_1 w_1 + x_2 w_2 + x_3 w_3 + b \qquad (2.2)$$

,де w_i - числові значення ваги. Ваги є значеннями, які будуть змінюватись протягом процесу навчання. b є вагою елемента зміщення на +1, включення ваги b робить вузол гнучкішим.

Вище було пояснено, як працює відповідний вузол/нейрон/персептрон. Але у повній нейронній мережі знаходиться багато таких взаємозв'язаних між собою вузлів. Структури таких мереж можуть приймати міріади різних форм, але найпоширеніша складається з вхідного шару, прихованого шару та вихідного шару. Приклад такої структури приведено нижче:

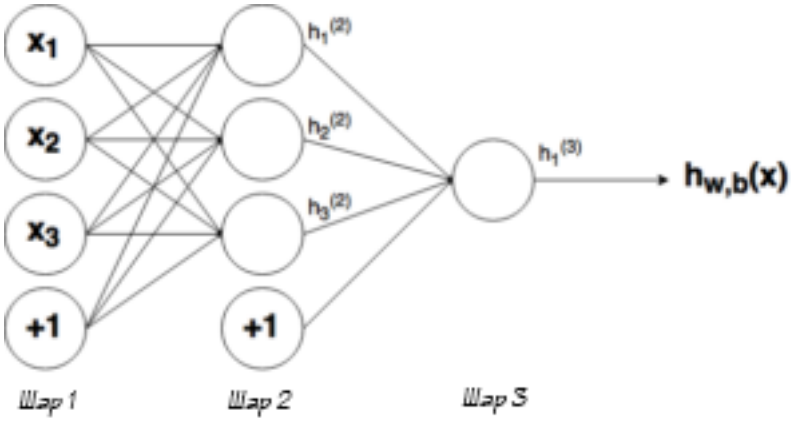

Рисунок 2.3 – Модель зв’язків в три-рівневій штучній нейронній мережі.

Ну рисунку вище можна побачити три шари мережі - Шар 1 є вхідним шаром, де мережа приймає зовнішні вхідні дані. Шар 2 називають прихованим шаром, цей шар не є частиною ні входу, ні виходу. Примітка: нейронні мережі можуть мати декілька прихованих шарів, у даному прикладі було включено лише один шар для простоти. І нарешті, Шар 3 є вихідним шаром. Ви можете помітити, що між Шаром 1(Ш1) та Шаром 2(Ш2) існує багато зв'язків. Кожен вузол у Ш1 має зв'язок зі всіма вузлами у Ш2, при цьому від кожного вузла у Ш2 йде по одному зв'язку до єдиного вихідного вузла у Ш3. Кожен з цих зв'язків повинен мати відповідну вагу.

Щоб продемонструвати, процес прямого поширення, маючи вже відомий вхід, у нейронних мережах, почнемо з попереднього прикладу з трьома шарами. Нижче така система представлена у вигляді системи рівнянь:

$$h_1^{(2)} = f(w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + w_{13}^{(1)} x_3 + b_1^{(1)}) \qquad (2.3)$$

$$h_2^{(2)} = f(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 + b_2^{(1)}) \quad (2.4)$$

$$h_3^{(2)} = f(w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3 + b_3^{(1)}) \quad (2.5)$$

$$h_1^{(3)} = f(w_{11}^{(2)}h_1^{(2)} + w_{12}^{(2)}h_2^{(2)} + w_{13}^{(2)}h_3^{(2)} + b_1^{(2)}) \quad (2.3)$$

Останній рядок розраховує вихід єдиного вузла в останньому третьому шарі, він є кінцевою вихідною точкою в нейронній мережі. У ньому замість зважених вхідних змінних (x_1,x_2,x_3) беруться зважені виходи вузлів з другого шару ($h_1^{(2)}$,$h_2^{(2)}$,$h_3^{(2)}$) та зміщення. Така система рівнянь також добре показує ієрархічну структуру нейронної мережі.

Наступним кроком підставляє до системи значення вагів та зміщень, та розраховуємо результат:

$$h_1^{(2)} = f(0.2 \times 1.5 + 0.2 \times 2.0 + 0.2 \times 3.0 + 0.8) = 0.8909 \quad (2.7)$$

$$h_2^{(2)} = f(0.4 \times 1.5 + 0.4 \times 2.0 + 0.4 \times 3.0 + 0.8) = 0.9677 \quad (2.8)$$

$$h_3^{(2)} = f(0.6 \times 1.5 + 0.6 \times 2.0 + 0.6 \times 3.0 + 0.8) = 0.9909 \quad (2.9)$$

$$h_1^{(3)} = f(0.5 \times 0.8909 + 0.5 \times 0.9677 + 0.5 \times 0.9909 + 0.2) = 0.8354 \quad (2.10)$$

Розрахунки значень ваг, які з'єднують шари у мережі, і є тим, що ми називаємо навчанням системи. У контрольованому навчанні ідея полягає у тому, щоб зменшити похибку між входом та потрібним виходом. Якщо ми маємо нейронну мережу з одним вихідним шаром та деякий вхід x і ми хочемо, щоб на виході було число 2, але мережа видає 5, то знаходження похибки виглядає як abs(2−5)=3. Говорячи мовою математики, ми знайшли норму помилки L1.

Сенс контрольованого навчання у тому, що надається багато пар вхід-вихід вже відомих даних і потрібно змінювати значення ваг, базуючись на цих прикладах, щоб значення помилки стала мінімальною. Ці пари входу-виходу

позначаються як (x(1),y(1)),…,(x(m),y(m)), де m є кількістю екземплярів для навчання. Кожне значення входу або виходу може представляти собою вектор значень, наприклад x(1) не обов'язково має лише одне значення, воно може містити N-вимірний набір значень.

У навчанні мережі, використовуючи (x,y), метою є покращувати її у знаходженні правильного y при відомому x. Це робиться через зміну значень ваг, щоб мінімізувати похибку, для цього знадобиться градієнтний спуск (рисунок).

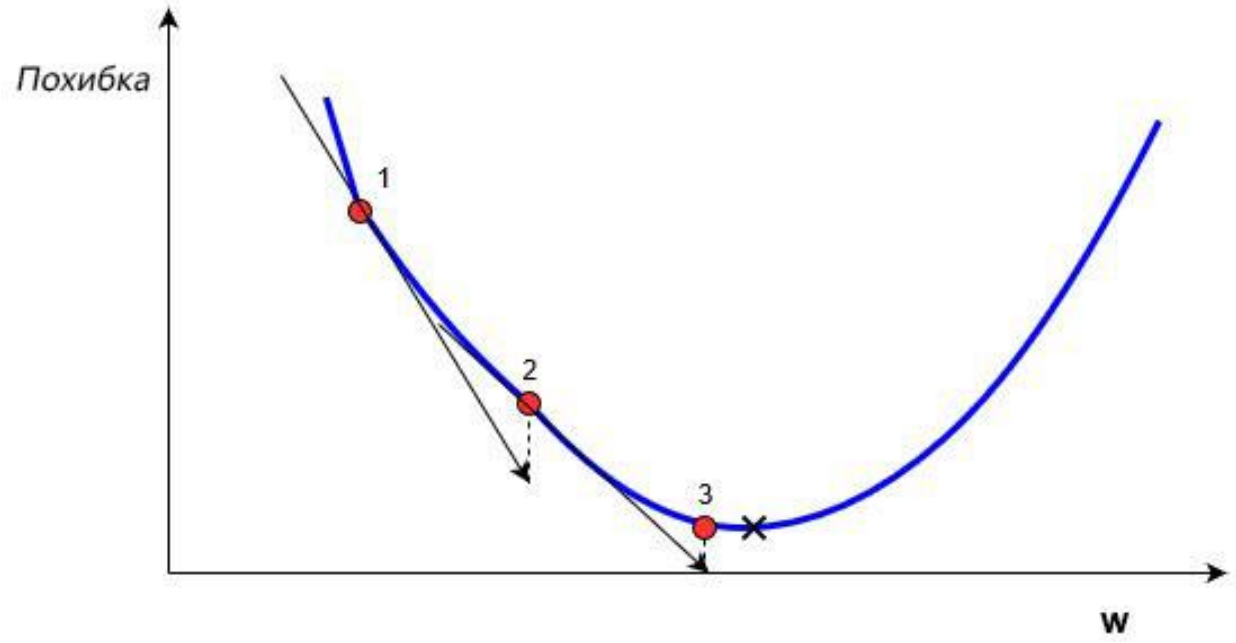

Рисунок 2.4 – Похибка в залежності від скалярного значення ваги w

На цьому графіку зображено похибку, залежну від скалярного значення ваги, w. Мінімально можлива похибка позначена чорним хрестом, але ми не знаємо яке саме значення w дає нам це мінімальне значення. Підрахунок починається з рандомного значення змінної w, яка дає похибку, позначену червоною крапкою під номером "1" на кривій. Нам потрібно змінити w таким чином, щоб досягнути мінімальної похибки, чорного хреста. Одним з найпоширеніших способів є градієнтний спуск.

Спочатку знаходиться градієнт похибки на "1" по відношенню до w. Градієнт є рівнем нахилу кривої у відповідній точці. Він зображений на графіку у вигляді чорних стрілок. Градієнт також дає деяку інформацію про напрямок - якщо він позитивний при збільшенні w, то у цьому напрямку похибка буде збільшуватись, якщо негативний - зменшуватись(див. графік). Як ви вже зрозуміли, ми намагаємося зробити, щоб похибка з кожним кроком зменшувалась. Величина градієнта означає те, як швидко крива похибки або функція змінюється у відповідній точці. Чим більше значення, тим швидше змінюється похибка у відповідній точці у залежності від w.

Метод градієнтного спуску використовує градієнт, щоб приймати рішення щодо наступної зміни у w для того, щоб досягнути мінімального значення кривої. Він є ітеративним методом, кожен раз оновлюється значення w через:

$$w_{\text{н}} = w_{\text{ст}} - \alpha * \nabla error \tag{2.12}$$

,де $w_{\text{н}}$ означає нове значення w, $w_{\text{ст}}$ - поточне або "старе" значення w, ∇error є градієнтом похибки на $w_{\text{ст}}$ та α є кроком.

Крок α також буде означати, як швидко відповідь наближається до мінімальної похибки. При кожній ітерації у такому алгоритмі градієнт повинен зменшуватись. З графіку вище ви можете помітити, що з кожним кроком градієнт "стихає". Як тільки відповідь досягне мінімального значення, ми виходимо із ітеративного процесу. Вихід можна реалізувати способом умови "якщо похибка менше деякого числа". Це число називають точністю.

Градієнтний спуск для кожної ваги $w_{(ij)}^{(l)}$ та зміщення $b_i^{(l)}$ у нейронній мережі виглядає наступним чином:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{\partial}{\partial w_{ij}^{(l)}} J(w,b) \tag{2.12}$$

$$b_{ij}^{(l)} = b_{ij}^{(l)} - \alpha \frac{\partial}{\partial b_{ij}^{(l)}} J(w,b) \tag{2.13}$$

Вираз вище фактично є аналогічним представленню градієнтного спуску: $w_{new} = w_{old} - \alpha * \nabla error$. Немає лише деяких позначень, але достатньо розуміти, що ліворуч розташовані нові значення, а праворуч - старі. Знову ж таки задіяно ітераційний процес для розрахунку ваг на кожній ітерації, але цього разу базуючись на функції оцінки J(w,b).

Значення $\frac{d}{dw_{ij}^{(1)}}$ та $\frac{d}{db_i^{(1)}}$ є частковими похідними функції оцінки, базуючись на значеннях ваги. Це означає, що кожний крок залежить від нахилу похибки/оцінки по відношенню до ваги. Похідна також має значення нахилу/градієнту. Звичайно, похідна позначається як $\frac{d}{dx}$. x у даному випадку є вектором, а це значить, що похідна теж буде вектором, який є градієнт кожного виміру x.

Розглянемо приклад стандартного двовимірного градієнтного спуску. Нижче представлено діаграму роботи двох ітеративних двовимірних градієнтних спусків:

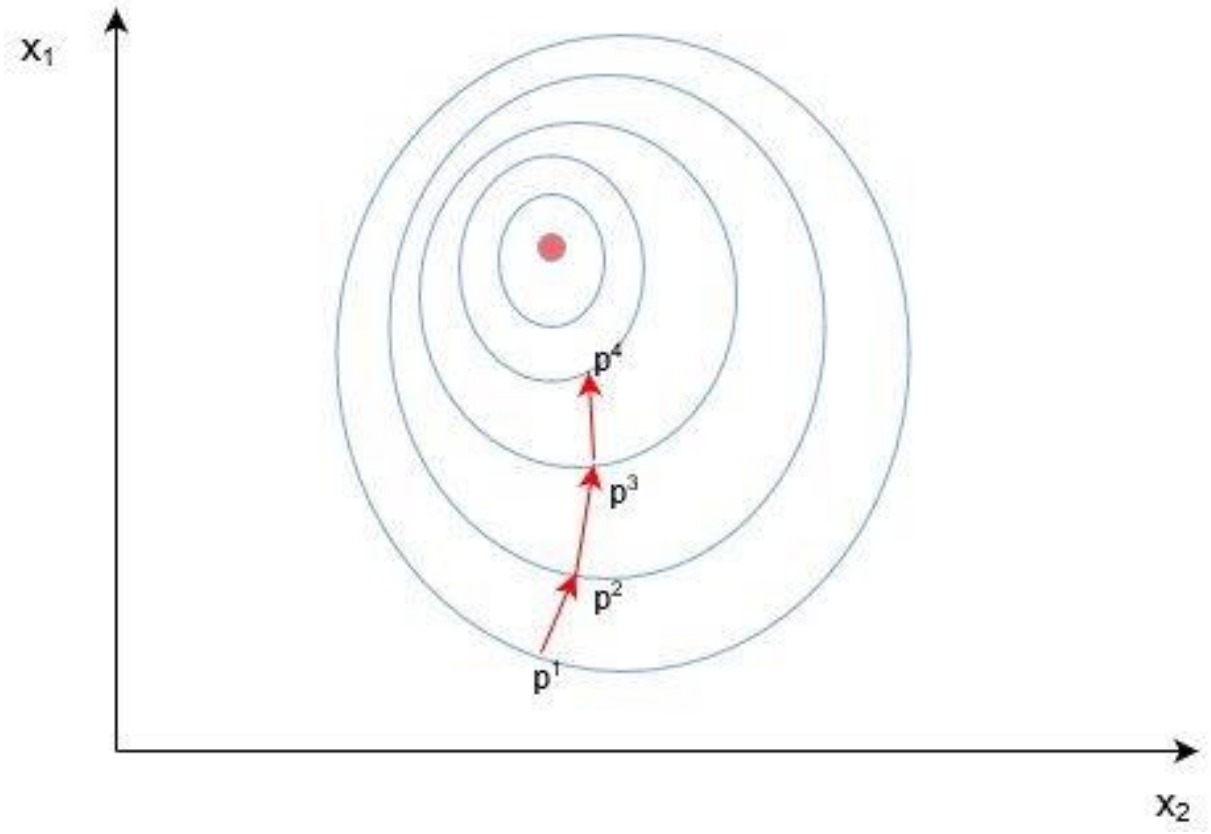

Рисунок 2.5 – Градієнтний спуск в двовимірному просторі

Синім позначені контури функції оцінки, вони позначають області, у яких значення похибки приблизно однакові. Кожен крок (p1→p2→p3) у градієнтному спуску використовує градієнт або похідну, що позначається стрілкою/вектором. Цей вектор проходить через два простори [x1,x2] і показує напрямок, у якому знаходиться мінімум. Наприклад, похідна, обчислена у p1 може бути $\frac{d}{dx} = [2.1,0.7]$, де похідна є вектором з двома значеннями. Часткова похідна $\frac{d}{dx_1}$ у цьому випадку дорівнює скаляру →[2.1], це є значення градієнта лише у одному вимірі пошукового простору (x_1).

Метод зворотного поширення дає можливість "ділити" функцію оцінки або похибку з усіма вагами у мережі. Іншими словами, можна з'ясувати, як впливає кожна вага на похибку.

Розглянувши математичну складову алгоритму зворотного поширення, отримаємо, що вихід цієї нейронної мережі знаходиться за формулою:

Після спрощення формула буде виглядати наступним чином:

$$h_1^{(3)} = f(w_{11}^{(2)} h_1^{(2)} + w_{12}^{(2)} h_2^{(2)} + w_{13}^{(2)} h_3^{(2)} + b_1^{(2)}) \tag{2.14}$$

$$z_1^{(2)} = w_{11}^{(2)} h_1^{(2)} + w_{12}^{(2)} h_2^{(2)} + w_{13}^{(2)} h_3^{(2)} + b_1^{(2)} \tag{2.15}$$

Для визначення як зміна конкретної ваги вплине на результат усієї функції потрібно визначити $\frac{dJ}{dw_{12}^{(2)}}$. Використавши правило диференціювання складної функції отримаємо:

$$\frac{\partial J}{\partial w_{12}^{(2)}} = \frac{\partial J}{\partial h_1^{(3)}} \frac{\partial h_1^{(3)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}} \tag{2.16}$$

Повний вигляд похідної функції оцінки:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b, x, y) = h_j^{(l)} \, \delta_i^{(l+1)} \tag{2.17}$$

2.2 Класифікація подій та наказів

Ігровий процес для ігор в жанрі стратегій в реальному часі є об’ємним і складним, оскільки в грі навіть двох гравців використовуються по меншій мірі десятки юнітів та будівель. Кожну секунду між сервером та клієнтами проходить величезна кількість даних, які потрібно обробити для використання для навчання та збереження даних. Умовно все, що відбувається під час ігрової сесії можна розділити на певні категорії та класифікувати.

Перший, і самий очевидний клас – накази. Накази – це основа управління в стратегіях. Замовлення військ, розміщення та побудова будівель, атака або відступ – все це накази гравців або агентів штучного інтелекту.

Другий клас, який в собі уособлює всі важливі події, які гравець або штучний інтелект не викликають наказами, це – події. Знищення будівлі, створення бойової одиниці, втрата стратегічної точки – всі ці події мають важливий вплив як на стратегічну складову, так і на тактичну, під час прийняття майбутніх рішень варто враховувати ці події.

Третій клас – події синхронізаційного рівня. Наприклад, бойова одиниця помітила ворога і почала наступати на нього або отримала пошкодження.

2.3 Штучний інтелект для мікроменеджменту

Для розробки штучного інтелекту буде застосовано штучні нейронні мережі, з певною специфікою, оскільки дані, які потрібно обробляти, мають комплексну структуру, та є нетиповими для використання в мережах.

Перш за все, для штучного інтелекту важливо враховувати правила тестового середовища. Тому агент буде використовувати доступні дані з певної зони – по суті те, що бачив би гравець. Отримані дані є набором

координат і характеристик військ. Для аналізу дані потрібно обробити і класифікувати.

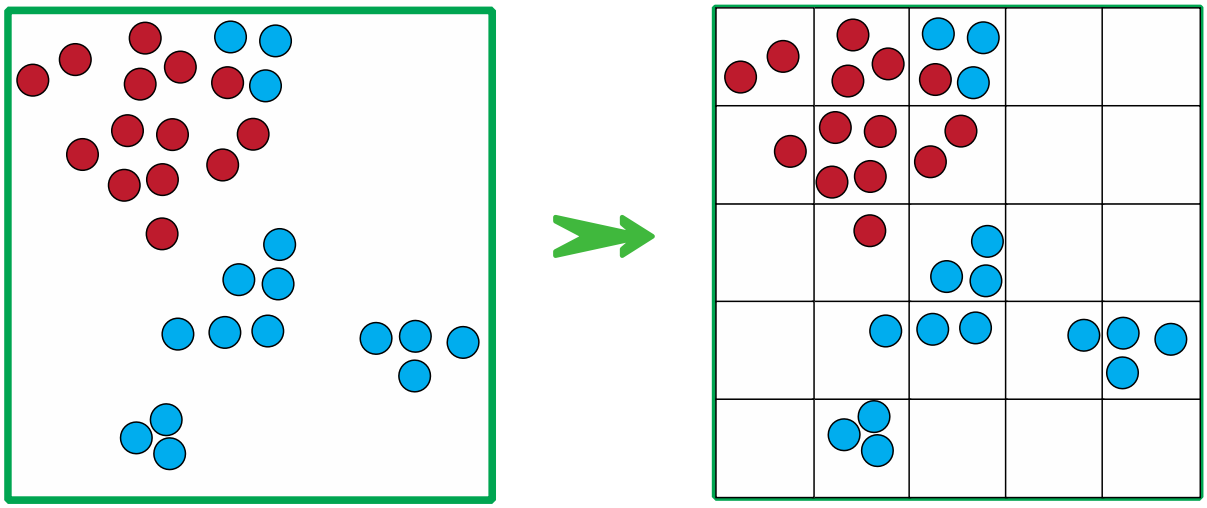

Рисунок 2.6 – Поділ території на зони

Отримані дані потрібно розбити на визначені невеликі частини. Управління – віддавання наказів – буде реалізовано для груп бойових одиниць, де кожен квадрат – окрема група (при відсутності військ накази не будуть віддаватись). Після розділу на групи умовна зона аналізу буде мати 25 під-зон, в кожній з яких буде окрема група під управлінням бота. Кожна під-зона має свій умовний номер, від 1 до 25, рахуючи зліва направо і зверху вниз.

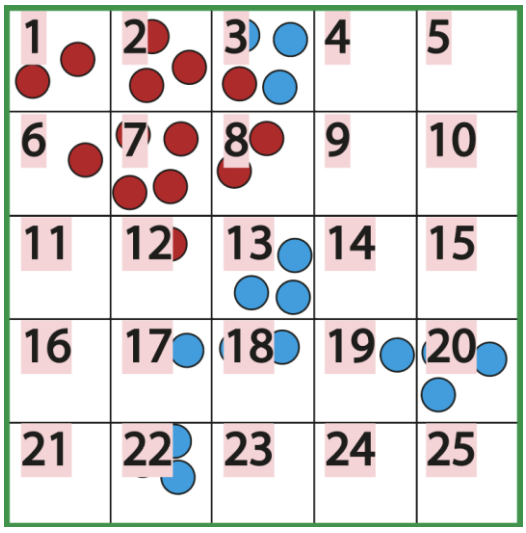

Рисунок 2.7 – Нумерація зон в межах визначеної території

Після попередньої обробки даних можна починати етап аналізу. Варто зауважити, що через агент буде аналізувати ситуацію та оновлювати накази через визначені періоди часу – середній час на виконання попередньої команди. Також варто зауважити, що управління кожною окремою групою буде проводитись в рамках 9 клітинок (рис. 2). Кінцева точка наказу для пересування групи буде в рамках «зони впливу» групи.

Візуалізація зон аналізу впливу і розміщення для 3 зони зображена на рисунку 2. Зона для аналізу позначена червоним, сусідні зони, що разом складають зону впливу, позначені червоною пунктирною лінією.

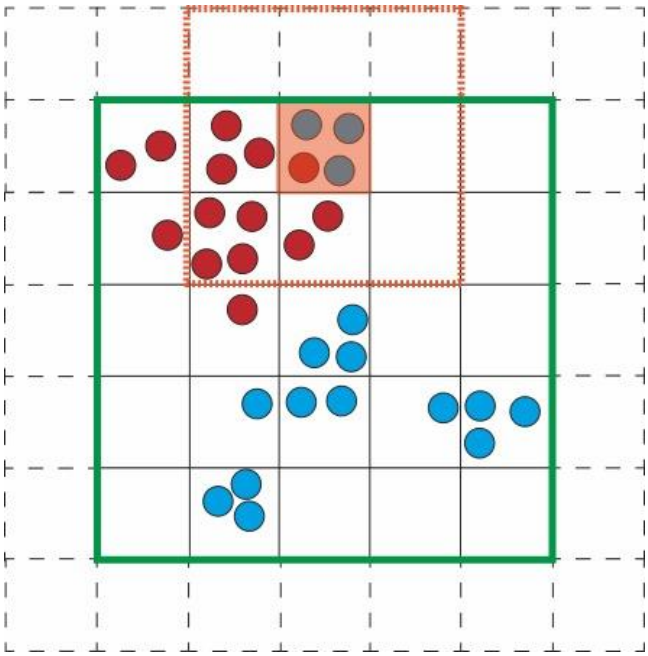

Рисунок 2.8 – Аналіз для зони номер 3

Мету роботи штучного інтелекту можна сформувати досить чітко – максимізація ефективності використання одиниць, що потрапляють в територію управління. Для метрики ефективності в зоні контролю агенту буде встановлено систему обрахунку втрат бойових одиниць – як супротивника, так і союзних. Основною метрикою системи буде вартість

одиниці, яка буде отримуватись з відповідної функції в кожного юніта в рамках тестового середовища.

Для прийняття рішення щодо подальших дій групи для кожної окремої зони буде проводитись 2 окремі дослідження – для аналізу зон впливу та аналізу ситуації в безпосередньому місці розміщення групи. На рисунку 3 можна побачити, як з карти території, розділеної на зони, ми отримуємо два результуючих набори даних, що є по суті матрицями результатів аналізу обох досліджень.

Фактичним результатами, з точки зору людини, будуть дані про можливість утримання поточної позиції виключно силами, які в ній знаходяться. Другий набір даних містить інформацію про позицію з урахуванням загальної кількості союзних та ворожих військ.

Вплив – поняття умовне, оскільки під ним мається а увазі можливість переходу в зону військ сусідніх зон за одиницю часу. Оскільки в тестовому середовищі всі юніти рухаються з однаковою швидкістю, одиниця часу – це період, за який група може перейти з поточної зони в будь-яку сусідню в режимі «рух до позиції» без урахування можливих непередбачуваних подій (наприклад, великої концентрації бойових одиниць супротивника або необхідності складного маршруту через перешкоди).

Комбінації цієї інформації достатньо для розуміння ситуації людиною та прийняття правильного рішення, виходячи з цього нейронна мережа також зможе зробити правильний вибір серед доступних варіантів. Для кожної зони є 9 варіантів руху з 3 можливими режимами переходу, всього 27 можливих варіантів для кожної групи.

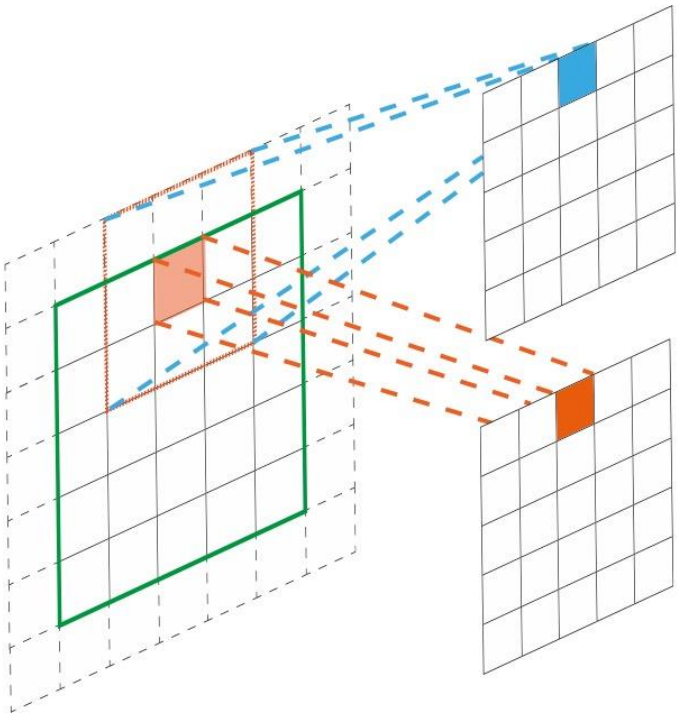

Рисунок 2.9 – Згортка карти для зон впливу (верхній, позначено синім кольором) та ситуації в поточній зоні (нижній, позначено оранжевим кольором)

На рисунку 3 також навколо території, що виділена зеленим, додано ряд зон. Необхідність в додаткових зонах полягає у єдиному підході для аналізу оточуючих зон. Оскільки для аналізу потрібно щоб у кожній зоні були сусіди з усіх сторін, було використано аналог техніки Padding, що часто використовується для аналізу зображень в згорткових нейронних мережах.

Після аналізу дані про оточуючі зони та поточний стан у конкретній зоні передається в нейронну мережу, яка в свою чергу надає на вихід дані про напрям руху (або утримання поточної зон) і режим руху. На рисунку 4 надано схематичне зображення структури обробки даних для однієї зони після проведення перших двох етапів аналізу.

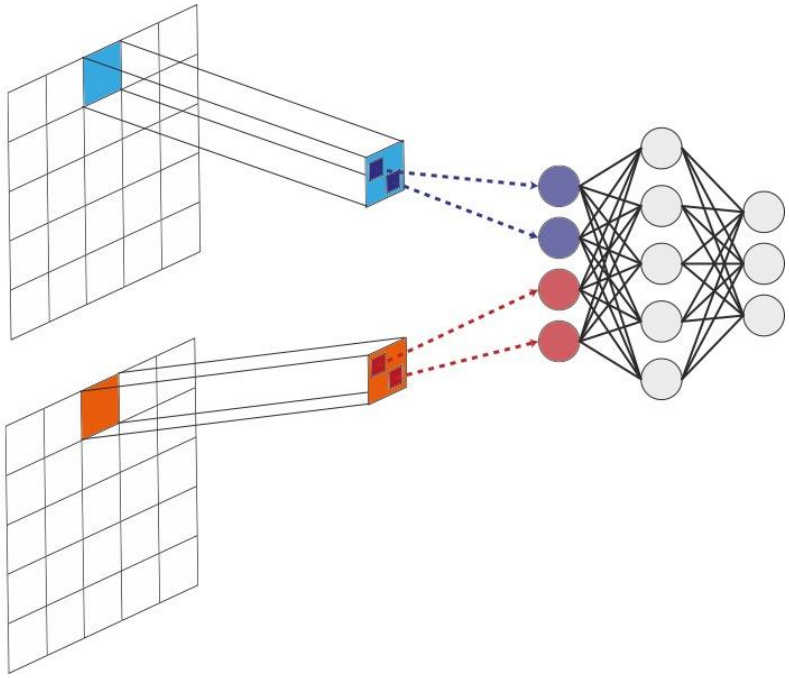

Рисунок 2.10 – Мережа для об’єднання результатів попередніх аналізів

Результатом для зони 3 є напрям, що можна виразити вектором [1;-1] і режимом переходу 0. Цей режим переходу означає наказ для групи одиниць на рух без зупинок не зважаючи на бойові одиниці супротивника, що будуть зустрічатись на шляху. Вектор можна зрозуміти як рух в праву нижню зону (або ж зону під номером 9).

На рисунку 5 кінцевий наказ, наданий групі в зоні 3. Відступ був необхідний через велику концентрацію одиниць супротивника, хоча в зоні перевага була за військами агента.

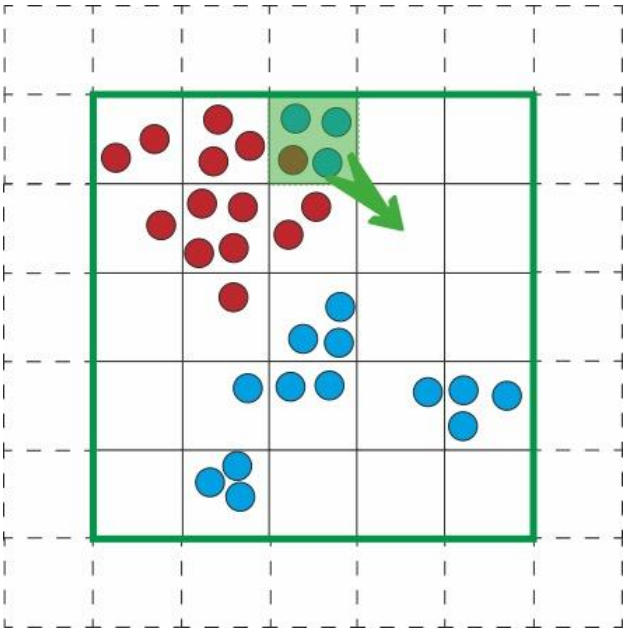

Рисунок 2.11 – Наказ групі в зоні 3

2.4 Стратегічний штучний інтелект

Для створення модуля штучного інтелекту, який буде відповідати за глобальне планування, або макроменеджмент, в стратегіях використовують різні підходи. Як було згадано в огляді існуючих рішень, для цієї частини штучного інтелекту часто використовують підходи, що розділяють всі здачі на набір дій, кожна з яких має певну «ціну». Задачею агенту є максимізація суми оцінок результатів цих дій.

Найпростішим варіантом для створення агенту є проста послідовність дій, що мають виконуватись послідовно в чітко визначеній послідовності. Такі системи часто і досить ефективно можна використовувати для початкової, або дебютної, фази гри.

Загальний термін «дебют» прийшов у стратегії з покрокових настільних ігор, найчастіше його використовують у шахах. В рамках покрокових ігор «дебютними» вважаються перші 10-15 ходів, однак ця кількість може залежати від типу гри. За цей час визначаються основні напрями дій гравця, а

також проводиться «мобілізація» та розміщення фігур або інших рухомих ігрових елементів на позиціях.

Для ігор в жанрі стратегій в реальному часі «дебютна» частина займає від 2 до 10 хвилин ігрового процесу, це залежить від специфіки гри. Також, через залежність від швидкості віддавання наказів гравцями, час дебюту у вмілих гравців менший. За цей час гравці визначають основні напрями свого майбутнього розвитку, це в свою чергу впливає на подальшу стратегію.

По загальним правилам в стратегіях з самого початку у кожного з гравців є досить велика кількість ресурсів, які потрібно правильно розподілити та використати зважаючи на стратегію. Розподілення цих ресурсів і визначає дебютну частину, що дозволяє у результаті визначити подальшу стратегію.

В рамках розроблюваного тестового середовища для дебюту можна використати наступні варіанти:

- Економічний;
- Технологічний;
- Мілітаристичний.

Кожен з варіантів має чітко визначений набір дій, що може бути формалізований у послідовну структуру. Структуру для «мілітаристичного» варіанту можна побачити на наступному рисунку:

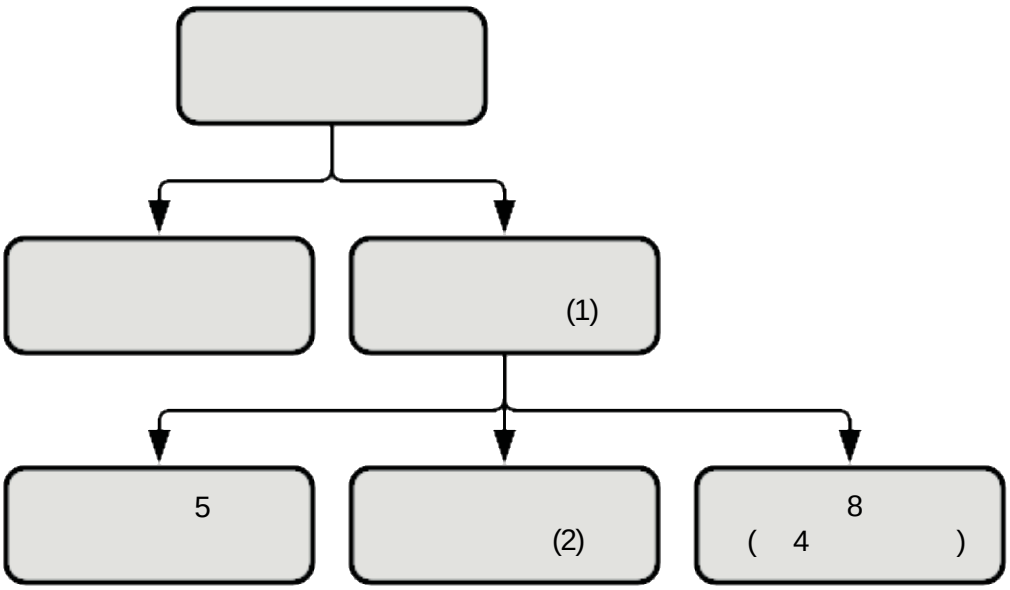

Рисунок 2.12 – Структура «мілітаристичного» варіанту початкової фази гри

Такий варіант розвитку, як можна побачити з рисунку, дозволяє почати агресивні бойові дії одразу після завершення початкового етапу гри – освоєння початкових ресурсів. На цей момент у гравця буде достатня ресурсна база для підтримання виробництва великої кількості низько-рівневих бойових одиниць, також після такого варіанта дебюту у гравця одразу буде потрібна кількість бойових одиниць для захоплення контрольних точок чи навіть атаки ворожої бази.

Економічний варіант дозволяє максимально наростити економічний потенціал гравця, маючи після дебютної фази велику ресурсну базу та велику кількість робітників. Технологічний варіант початкової фази дозволяє збалансувати економічну та мілітаристичну складову, для отримання доступу до самих важких юнітів вже після початкової фази.

Кожен з варіантів початкових дій окрім очевидних переваг має і свої недоліки. Для економічного варіанту основною проблемою є відсутність можливостей до швидкого набору бойових одиниць та відсутності будь-якого

захисту. Технологічний варіант не надає необхідного рівня розвитку економіки для повноцінного використання всіх переваг важких бойових одиниць. Мілітаристичний варіант важко реалізувати на довгострокову перспективу, адже після початкової фази він має недостатній розвитку як технологій, так і економіки.

Для адаптації різних початкових варіантів розвитку до ефективної подальшої гри можна використати ієрархічну мережу задач. Загальний вигляд такої мережі наступний:

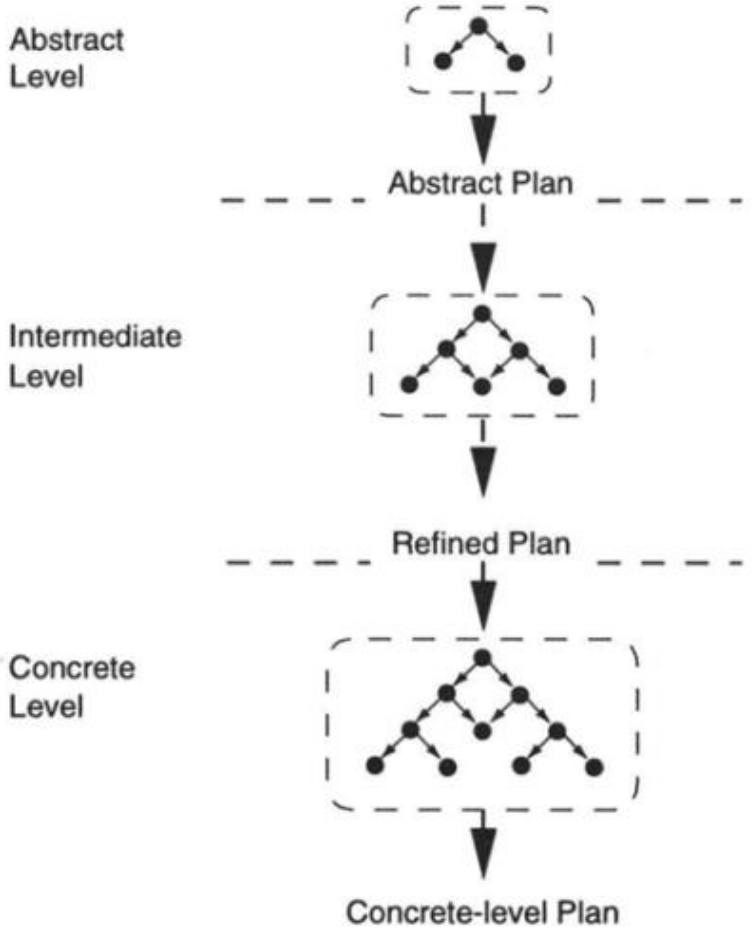

Рисунок 2.13 – Три-рівнева структура ієрархічної системи планування

Така мережа будується за наступними правилами. Верхній рівень сприймає задачу високого рівня – «перемога над супротивником (ами)».

Наступним етапом є розділення даної задачі на під-задачі. Для стратегій наступний набір може різнитись в залежності від режиму, наведений набір актуальний для перемоги в режимі «ліквідація»:

- Контроль над ресурсними точками;
- Знищення робітників;
- Знищення ресурсної бази;
- Знищення центру бази.

Вказані вище задачі можуть бути поставлені одночасно, однак їх виконання повинно слідувати певній пріоритетності. Кожна з цих задач може вирішитись певним комплексом дій, який необхідно застосувати.

Останній рівень мережі – постановка конкретних задач. У випадку показаної схеми ситуація, для якої створена конкретна мережа, не відображає загальні можливості ієрархічних мереж завдань. Хоча структурно всі вони мають подібну схему, до рівня отримання конкретних завдань може бути багато рівнів розділення задач на під-задачі.

В рамках стратегії виконання під-задачі основної цілі («перемоги») «контроль позиції» розділяється на цілий набір завдань. Наступний набір задач буде надано до виконання:

- Напад одиницями що є в наявності для розвідки зони;
- Якщо зона перейшла під контроль – посилити захист, в іншому випадку – створення нових одиниць;
- Якщо контроль над зоною не вдається встановити – покращити виробничу частину для отримання можливості активного направлення підкріплень;
- При недостатній кількості ресурсів – збільшити економічну базу;
- При наявності у супротивника критичної переваги у технологічному рівні – підвищити власний рівень технологій.

Даний набір задач не є прямими наказами, кожна з них розділяється на ще один невеликий набір завдань, кожне з яких уже можна розділити на

конкретні накази, що можна надати до виконання ігровим одиницям або будівлям.

Основна ціль ієрархічної мережі задач – рекурсивна декомпозиція задач до рівня наказів у рамках тестового середовища [27]. Такі мережі для планування використовуються у багатьох сферах і показують свою досить високу ефективність у порівнянні з іншими методами. На наступному рисунку можна побачити приклад розділення задач на конкретні накази.

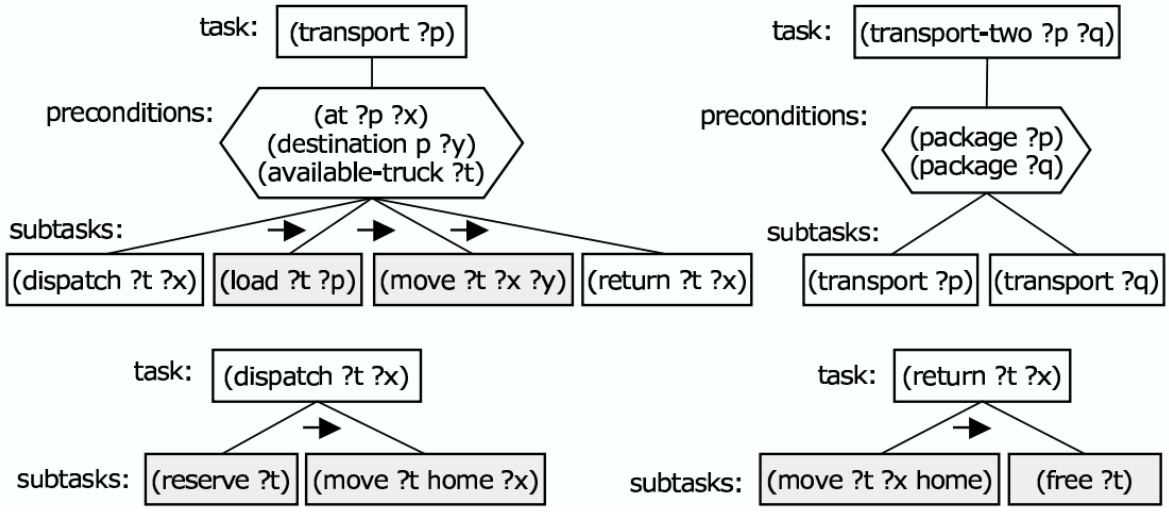

Рисунок 2.14 – Розділення задачі переміщення товарів на конкретні кроки (p, q – пакунки для доставки; t – транспорт; x, y – точки транспортування)

Як видно з рисунку, задачею, що була поставлена перед системою, є перевезення одного або декількох товарів з однієї точки до іншої. Постановка кожної з задач розділяється на набір умовних рівнів, які відповідають за різні елементи постановки завдання:

- Перевірка виконання умов;
- Розділення на під-задачі;
- Декомпозиція під-задач до конкретних наказів/завдань.

Як можна побачити з рисунку, така тривіальна задача як доставка двох пакунків з точка А в точку Б вибудовується в наступну мережу завдань:

1. Поставлена задача проходить перевірку на валідність, в випадку задачі доставки двох товарів перевіряється чи товари доступні в точці А;

2. За умови проходження перевірки задача розбивається на дві під-задачі, кожну з яких є завданням переміщення 1 пакунку;

3. Кожна з під-задач переміщення одного пакунку є типовою, тому розглядатимемо переміщення пакунку під номером 1. Варто зауважити що в даному випадку логістична система враховує правило переміщення одного пакунку одним транспортом. Тобто, під одним пакунком в даній системі розробники розуміють кількість або об’єм товару, що повністю заповнює транспорт. В інших логістичних системах можлива більш складна ієрархічна структура, коли на переміщення потрібен набір товарів, а логістична складова – маршрут та кількість потрібного транспорту – залежить від мережі завдань. В такому випадку перед самим відправленням формується задача на комбінування товарів у максимально малу кількість пакунків для оптимізації процесу переміщення;

4. Для переміщення товару на складі, де він знаходиться, потрібен вільний транспорт. Тому першим завданням перед початком переміщення товарів буде отримання вільного транспорту;

5. Після того як транспорт був отриманий та направлений на склад, де зберігається пакунок, транспорт потрібно розвантажити для звільнення місця, цей етап також можна назвати підготовкою до транспортування;

6. Розвантаживши та підготувавши транспорт до перевезення нової партії товару, завантажити пакунком;

7. Завантаживши пакунок транспорту надається наказ переміщення до точки призначення;

8. Після прибуття в точку та розгрузки пакунку транспорт отримує завдання повертатись додому.

Повноцінний розбір завдання про транспортування пакунків можна побачити на рисунку 2.15.

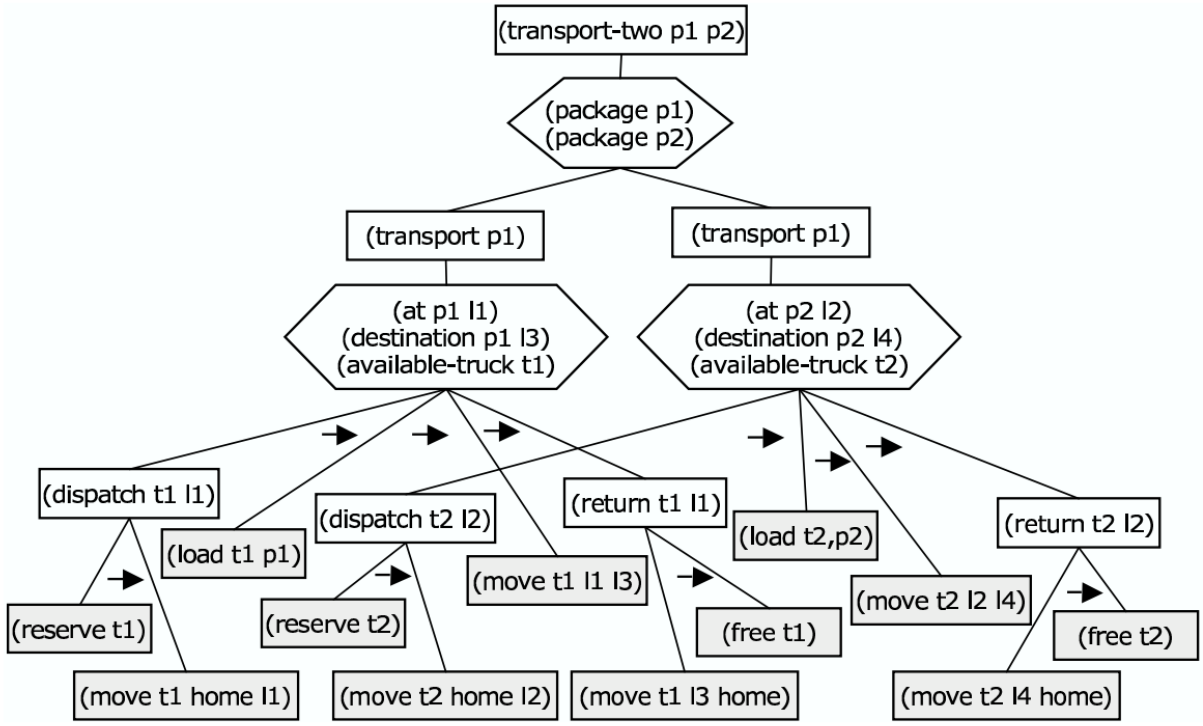

Рисунок 2.15 – Детальний опис послідовності дій

Для стратегій такий порядок виставлення завдань є типовим, оскільки складні задачі просто необхідно розділити до більше простих завдань, що в кінцевому випадку зводяться до виконання наказів.

Конкретно ця структура показує роботу між ринками гравців під час гри. Для створення додаткових механік в стратегічних іграх часто застосовують механізм торговців, які взаємодіють з ринками інших гравців або з стаціонарними точками. Такий обмін товарами дозволяє отримувати додаткові ресурси, а також вводить для супротивників додаткову можливість впливу на економіку.

Така структура ефективно може працювати з більшістю задач. Окрім розділення завдань на накази проблема в стратегіях полягає у необхідності вибору оптимального варіанту дій. Наприклад, розвиток економіки зараз може відстрочити створення нової групи бойових одиниць, але в майбутньому принесе користь через збільшення потоку отримуваних ресурсів. Через це

правильна пріоритизація завдань для ієрархічної структури залишається одним з ключових факторів, що визначає загальну ефективність.

2.5 Висновки до розділу

Для створення тактичного штучного інтелекту було обрано рішення з використанням нейронних мереж. В порівнянні з попередніми дослідженнями новими елементами є застосування попереднього аналізу для визначення зон впливу та реальної кількості одиниць, використання підходу, аналогічного з Padding, що ефективно використовується в згорткових нейронних мережах.

Для стратегічного штучного інтелекту була обрана комбінація з «типовою» стартовою послідовністю дій і подальшим використанням ієрархічної мережі завдань для досягнення ефективного стратегічного управління в пізніх етапах гри.

Використання саме таких підходів дозволить створити гнучкий, потужний та відносно економний штучний інтелект, що зможе показувати ефективну роботу навіть на мобільних пристроях.

3 ПРОЕКТУВАННЯ СЕРЕДОВИЩА ТЕСТУВАННЯ

3.1 Вибір ігрового рушія

Для отримання реальних результатів та визначення ефективності спроектованого та розробленого штучного інтелекту потрібне тестове середовище. Особливо важливою є розробка тестового середовища для ігор в жанрі стратегій в реальному часі.

Особливістю ігор в даному жанрі є використання специфічного набору механік. Наприклад, в деяких представниках жанру основною механікою є утримання територій. Ресурсна база залежить від контрольованих зон на карті, а за деякі зони гравець отримує бали, необхідні для перемоги. Це спричиняє необхідність агресивної гри, оскільки втрата територій автоматично означає втрату ресурсів, що несе за собою програш.

На противагу стратегіям, де основною механікою виступає заволодіння та подальший захист територій, класичні стратегії базуються на складній економічній системі. Окрім бойових будівель гравець може створювати ресурсні. В грі часто є більш ніж один ресурс, тому гравцю, окрім того, що треба створювати економічні будівлі і забезпечувати їх працівниками, потрібно ще й контролювати і балансувати ресурси згідно з потребами і обраною стратегією.

Таких особливостей можна знайти багато. В деяких стратегіях ключовим є управління кожною бойовою одиницею, для деяких – створення сильної економіки і використання передових технологій, інші використовують механіку величезних армій.

Від усіх вищезазначених факторів будуть залежати результати штучного інтелекту, а також кожної окремо виділеної його частини.

Для тестування комплексного штучного інтелекту найоптимальніше буде створити прототип типової стратегії з деякими спрощеннями. Система повинна враховувати можливості гри на мобільному пристрої, також ця

метрика є важливою для тестування. Комплексний штучний інтелект є досить складним для використання на пристроях користувачів, оскільки смартфони та планшети наразі не можуть повноцінно конкурувати з потужностями комп'ютерної техніки, а особливість штучного інтелекту в іграх полягає в необхідності його постійної роботи – обробки даних та прийнятті нових рішень.

Для тестування штучного інтелекту використовують ігровий рушій з назвою microRTS. Це мінімалістичний та спрощений варіант стратегії в реальному часі з обмеженими можливостями, однак є досить популярним та цікавим варіантом для перевірки штучного інтелекту власної розробки з штучним інтелектом інших авторів. Загалом і рушії використовуються декілька варіантів бойових юнітів, вони відрізняються кількістю одиниць здоров'я, силою та типом (розподіл на далекобійні та рукопашні бойові одиниці). Перелік доступних ігрових об'єктів наступний:

- Робоча одиниця;
- Легка рукопашна бойова одиниця;
- Важка рукопашна бойова одиниця;
- Далекобійна бойова одиниця;
- Казарми;
- Мінерали (добуваються робочими одиницями, єдиний ресурс).

Можливості в microRTS обмежені, однак це не заважає їй бути якісним замінником тестового середовища для тестування штучного інтелекту. Однак варто зважати на те, що в готовому середовищі досить складно щось змінювати, а готові варіанти штучного інтелекту не будуть працювати з новими типами бойових одиниць чи будівель. Також важливою відмінністю від звичайних стратегій є відсутність технологічного прогресу. На рисунку зображено ігрове полу в microRTS з бойовими одиницями та певним набором будівель.

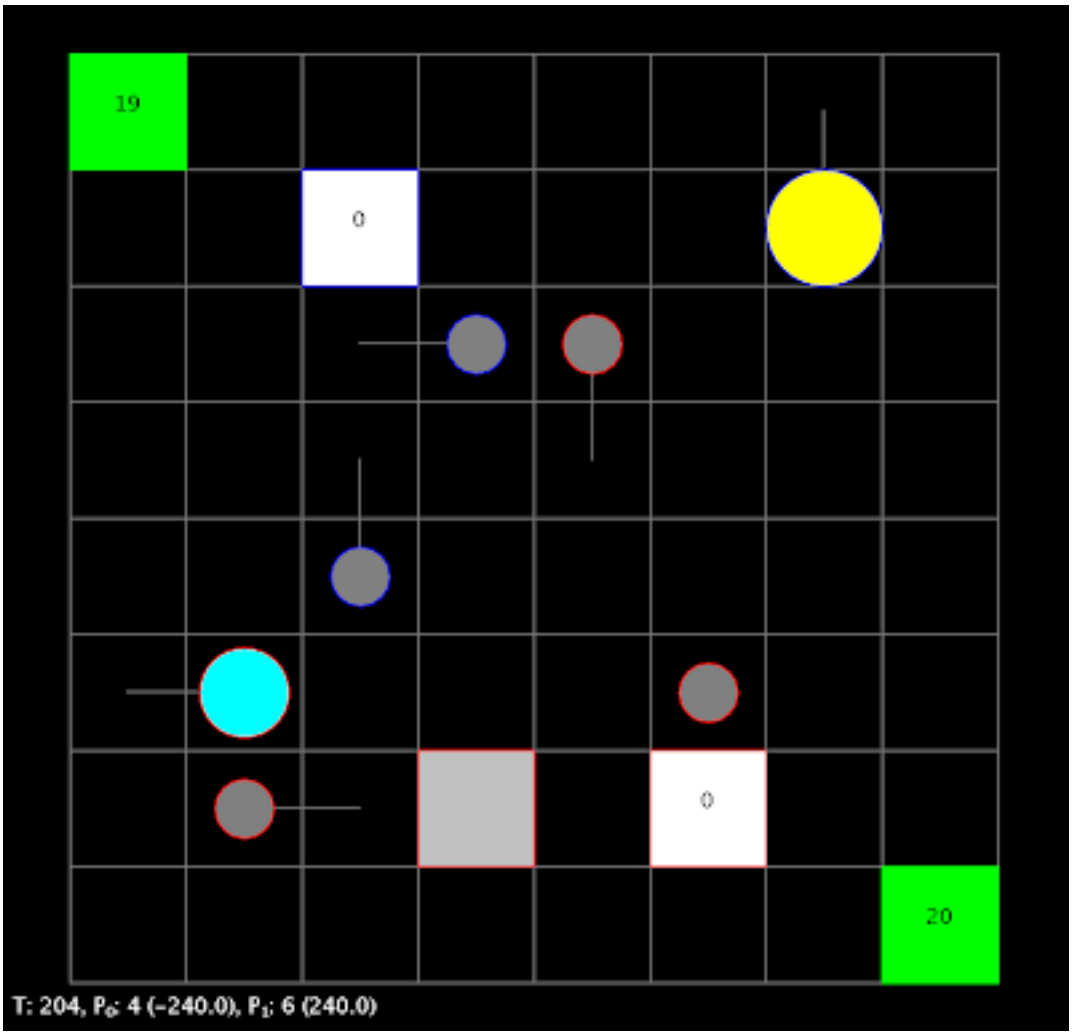

Рисунок 3.1 – Ігрове поле під час гри в microRTS

Враховуючи всі визначені вище особливості та функції, які має підтримувати ігровий рушій, microRTS не є підходящим рішенням. Тому варіантами для вибору стають популярні зараз ігрові рушії, такі як Unreal Engine, Unity, Godot та інші. Написання ігрового рушія самостійно є неоптимальним рішенням з точки зору витрат часу на весь функціонал ігрового рушія.

Отже, основними рушіями, які користуються популярністю і розробників ігрових застосунків є Unreal Engine та Unity. Інші ігрові рушії є не такими популярними, а для створення тестового прототипу не потрібно враховувати всі особливості роботи застосунку на пристрої користувача.

Ключовими моментами, найважливішими для створення прототипу та подальшого тестування на його основі, є:

- Швидкість розробки;
- Активність користувачів ігрового рушія;
- Крива вивчення;
- Наповненість різнотипними ассетами колекцій у офіційних магазинах;
- Можливість легкого порту на мобільні телефони;
- Підтримка різних протоколів для взаємодії з сервером.

Для порівняння наведемо список відмінностей обох ігрових рушіїв [28]:

- Unreal Engine є відкритим для змін ігровим рушієм, Unity має закриту кодову базу.
- Створення: Unreal Engine дебютував в 1998 році, Unity був анонсований і випущений в 2005 році.
- Мови: Unreal engine використовує C++ і Unity використовує C#.
- Спільнота: Обидва рушії мають величезну активну спільноту. Тим не менш, Unity 3D має набагато нижчий поріг для входу ніж Unreal 4, а також він має більше клієнтів і, отже, базу користувачів.
- Документація: Обидва пропонують хорошу і детальну документацію з поясненням їх інструментів і особливостей. Тим не менш, Udemy пропонує більш широкий спектр курсів Unity.
- Asset Store: Магазин ассетів дозволяє користувачам завантажувати такі елменети, як текстури та моделі. Unity має ширший спектр модифікацій у порівнянні з Unreal. Unreal має близько 10000 ассетів, тоді як Unity має 31000 ассетів. Розглядаючи цю відмінність варто зауважити, що ассети для Unity часто несумісні та не підтримують стандартизацію, що вносить додаткові проблеми в проекти.
- Графіка: Обидва інструменти мають хорошу графіку, але Unreal Engine переважає Unity через якість графіки.

- Вихідний код: Unreal Engine має відкритий вихідний код, що полегшує процес розробки. Unity, з іншого боку, не надає відкритого вихідного коду; однак його можна купити.
- Рендеринг: Unreal підтримує швидшу візуалізацію, роблячи пост-обробку ще швидшою. Рендеринг є повільним у випадку єдності, отже обробка проектів також є повільною.
- Ціноутворення: Unreal Engine поставляється безкоштовно, але ви володієте роялті до них. Unity доступний безкоштовно, але повна версія може бути оновлена з одноразовою оплатою $ 1500 або $ 75 / місяць.

Розглянувши дані інформацію можна зробити висновок що рушії мають свої переваги та недоліки. Для максимального спрощення по ключовим моментам дані буде записано в таблицю і розглянуто окремо.

Таблиця 3.1 – Порівняння Unreal Engine та Unity

Ігровий рушій / Параметри	Unreal Engine	Unity
Визначення	Рушій з відкритим кодом	Кросплатформенний ігровий рушій.
Розробник	Epic Games	Unity Technologies
Мови програмування	C++	C#
Сфера активного використання	Використовується для розробки ігор для ПК, мобільних телефонів. консолей тощо.	Використовується для розробки ігор для ПК, мобільних телефонів. консолей тощо.

Продовження таблиці 3.1

Найбільш успішні функції	Надійна багатокористувацька структура, VFX та моделювання частинок.	Підтримка 2D, якісні компоненти для створення анімації.
Вихідний код рушія	Вихідний код відкритий.	Вихідний код закритий
Ціноутворення	Безкоштовний до публікації комерційного продукту.	Базова версія безкоштовна.
Складність розробки	Високі поріг входу, потребує специфічних знань в багатьох сферах	Рушій має інтуїтивно зрозумілий інтерфейс та орієнтований на низький рівень входу.
Графікчні можливості	Доступний високий рівень графічних елементів, один з передових рушіїв у створенні фото-реалістичної графіки.	Надає хороші графічні можливості, однак досягти ефекту реалістичності складно

Проаналізувавши отримані дані, можемо зробити ряд наступних висновків:

- Unreal Engine підтримує вищі можливості для створення якісної графіки, ніж Unity.
- В Unreal Engine більше компонентів системи для розробки, частина логіки може бути створена за допомогою програм для візуального програмування.

- В Unity простіша крива навчання, також в роботі з ним потрібне знання меншої кількості інструментів.
- Обидва є кросплатформенними, однак для Unity більша частина функціоналу для переносу на різні платформи є вбудованою в код рушія та створена розробниками.
- Unity вважається ідеальним рушієм для створення та тестування прототипів.

Враховуючи всі вищезазначені причини, розробка тестового середовища буде виконуватись з використанням рушія Unity.

3.2 Створення проекту та основні налаштування

Для розробки проекту буде використано режим 3D гри (рисунок). Це дозволить в майбутньому розвинути прототип до повноцінного продукту, який буде цікавим для користувачів.

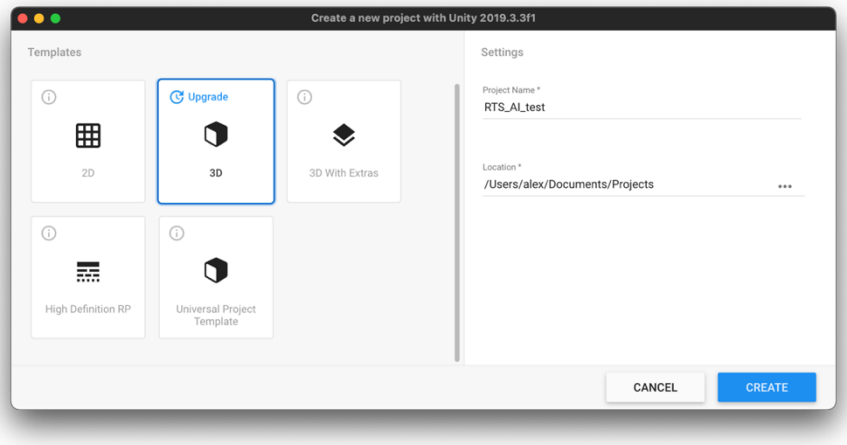

Рисунок 3.2 – Вікно створення проекту в Unity

Після створення потрібно налаштувати такі базові речі як камера та Skybox. Оскільки при розробці прототипу основна увага буде приділятись саме функціоналу, а не графіці, для проекту буде використовуватись спрощена система освітлення та роботи з тінями.

Для освітлення сцени буде використовуватись компонент Directional Light. Розміщений вище камери він буде давати рівномірне освітлення для всієї сцени.

Другим важливим і базовим елементом для стратегій є камер. На відміну від ігор в жанрі шутер або рольова гра, в стратегіях камера не має чіткого об’єкту для прив’язки. Оскільки користувачу постійно треба переходити до різних зон для управління дія, віддавання наказів та реагування на події, камера стає одним з базових елементів, що дозволяють виконувати ці дії.

Через простоту стратегії та мінімальну увагу до графічних особливостей, для камери буде використано компонент ортографічної (Orthographic). Такий вид камери не буде використовувати перспективу (рисунок), та для простої графіки це буде перевагою.

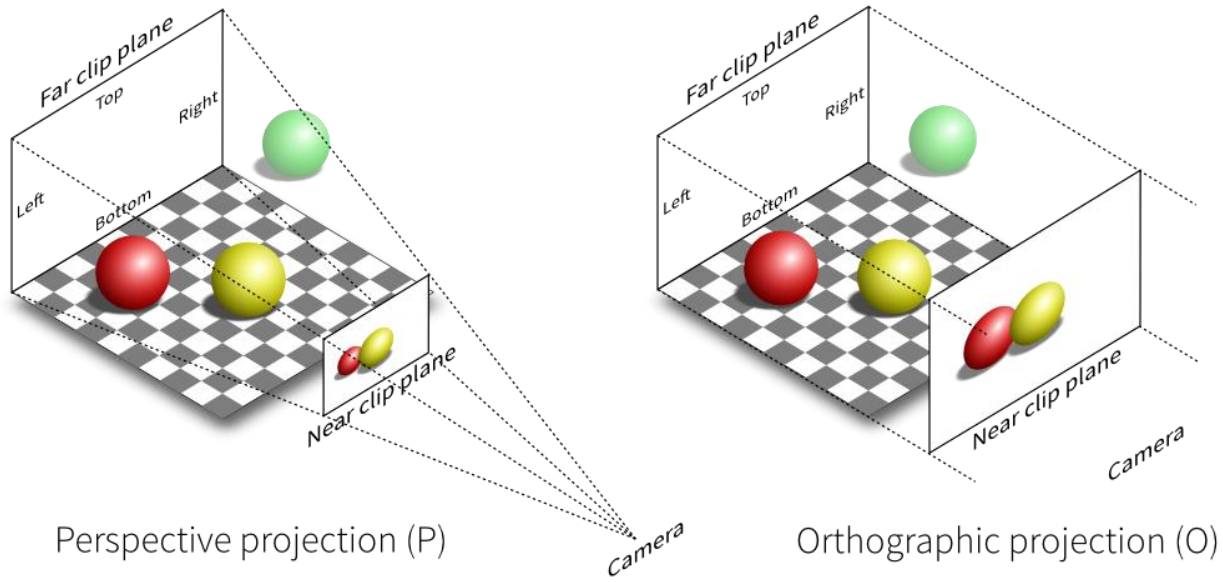

Рисунок 3.3 – Різниця перспективної (P) та ортографічної (O) проекцій

Камера буде мати декілька основних функціональних можливостей, що забезпечать ефективність користування даним інструментом для гравців:

1. Рух по периметру ігрової карти;
2. Наближення або віддалення до об’єктів;

3. Обертання навколо точки для отримання картинки ситуації з іншого ракурсу. Приклад такого переміщення можна побачити на рисунку, коли камера змінює свою позицію з точки C1 до точки C2 при цьому не змінюючи фокусу на об’єкті в точці P.

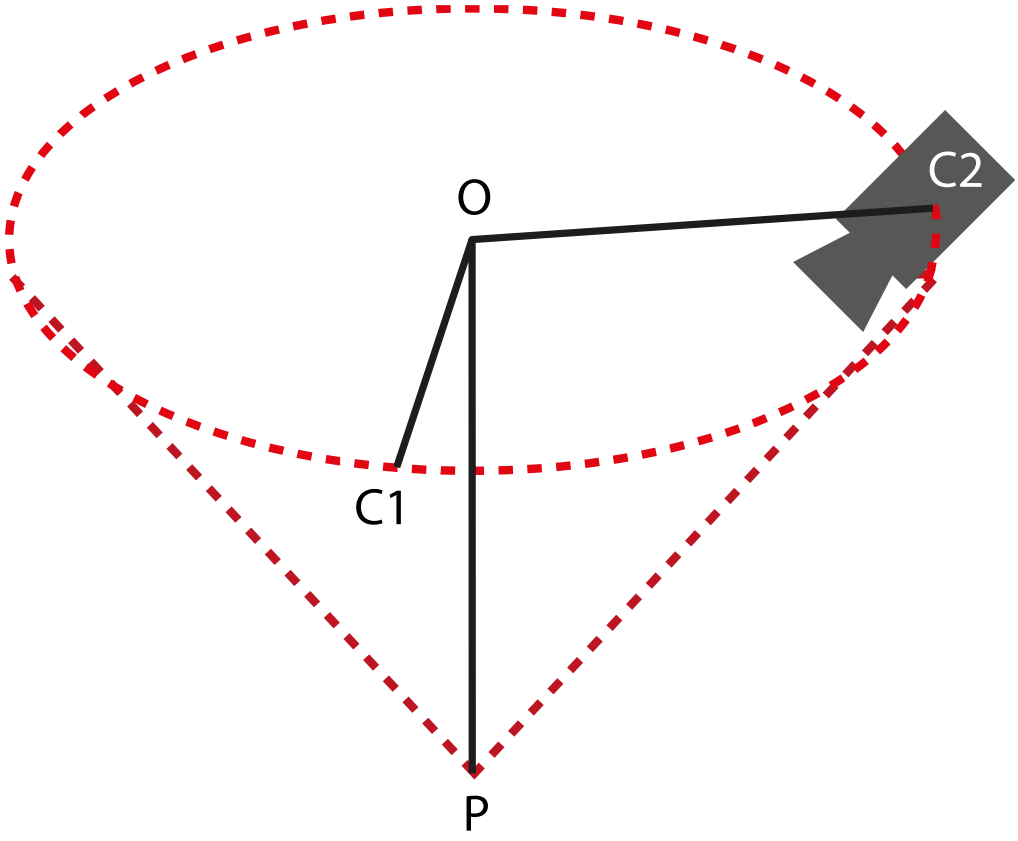

Рисунок 3.4 – Приклад повороту камери з точки C1 до точки C2 відносно центра (O) з напрямом на точку ігрового поля P, висота камери над полем – OP, кут нахилу – OC_2P

Рух камери по периметру є не надто складним завданням. Сама камера буде зберігатись в певній структурі об’єктів, кожен з яких буде відповідати за певний вид руху камери. Базовий об’єкт буде зберігати управління камерою, а також показувати її поточне положення. Саме за допомогою зміни координат даного об’єкта камера буде змінювати своє положення.

Для руху камери в коді буде вказано обробник користувацьких дій. Якщо користувач доторкнеться одним пальцем до екрану в змінну одразу буде

записано позицію, в якій він доторкнувся. Для цього з камери буде направлено луч безкінечної довжини, точка перетину ігрового поля з лучем і буде точкою дотику. Далі поки користувач не забере свій палець з ігрового поля камера буде змінювати свою позицію відповідно до рухів користувача, пропорційно змінюючи координати X та Y.

Для руху камери навколо певної точки потрібно використати декілька напрямів руху камери. Для такого руху користувачу потрібно двома пальцями зробити коловий рух по екрану. Камера змінить свою позицію пропорційно до відстані, на яку зміняться координати точки перетину луча та карти.

Однак особливістю саме такого варіанту управління є зміна також і кута нахилу камери таким чином, щоб вона дозволяла побачити точку, яку було видно в центрі камери в попередньому положенні.

Оскільки камера нахилена відносно осі X на 45 градусів, додатково потрібно змінити нахил по осі У. На рисунку показано приклад зміни положення камери з позиції С1 до позиції С2. Фокус камери повинен завжди знаходитись на точці Р, тому при переміщенні камери по колу потрібно змінювати кут нахилу по осі Y (в системі координат Unity Y – вертикальна вісь). Приклад результатів такого повороту камери можна побачити на рисунку 3.5.

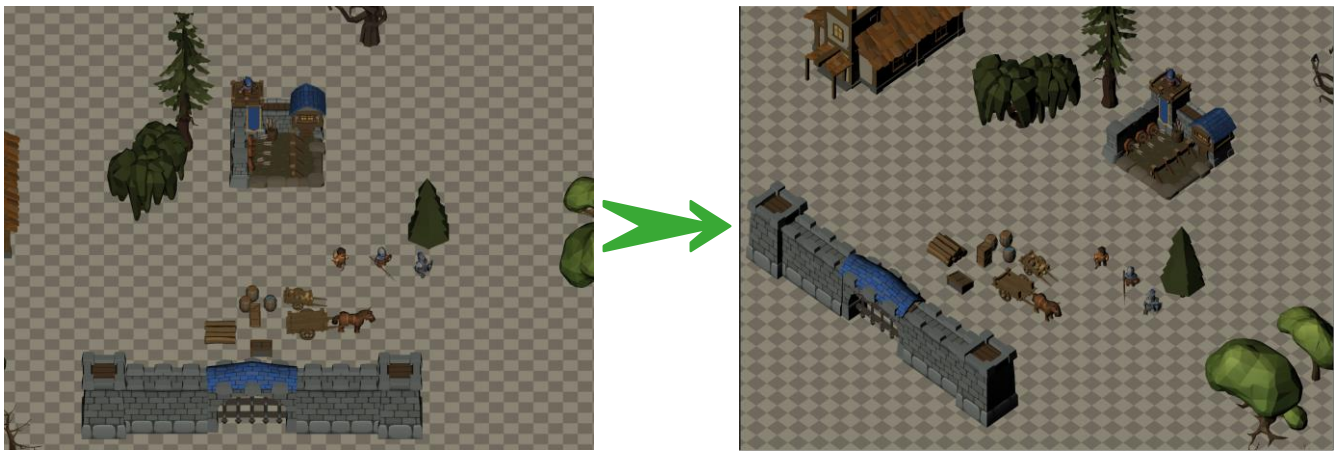

Рисунок 3.5 – Результати повороту камери відносно тестової сцени

3.3 Наповнення тестової сцени об’єктами

Головним об’єктом, на якому будуть відбуватись дії, є Terrain, розміщений в центрі сцени. Для тестового варіанту гри було обрано малий розмір мапи – 500 на 500 одиниць. Оскільки можливості симуляції фізики для тестового середовища штучного інтелекту стратегії не потрібні, Terrain не має специфічного матеріалу.

В рамках Terrain розміщуються всі внутрішні об’єкти в грі. Окрім будівель та інших бойових одиниць, на карті буде розміщено додаткові об’єкти, які є непрохідними для бойових одиниць гравця.

Першим елементом будуть дерева – бойовим одиницям доведеться їх уникати або обходити. Для створення дерев було використано моделі з безплатних ігрових елементів в колекціях Unity Store, для генерації дерев було використано вбудовану в компонент Terrain функцію по масовій генерації об’єктів (рисунок).

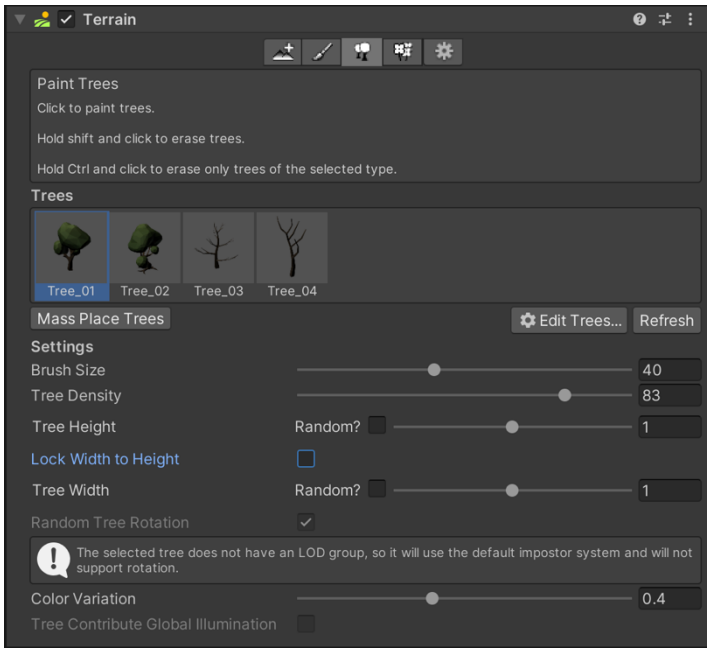

Рисунок 3.6 – Вікно розміщення дерев на елементі Terrain в Unity

Другим важливими елементом, який буде впливати на тактичну та стратегічну складові будуть непрохідні зони. Вони будуть складатись з різких перепадів висоти Terrain, та будуть недоступними для бойових одиниці. За допомогою даних зон ігрова карта буде суттєво обмежена, також з їх допомогою можна керувати основним напрямком руху військ.

На рисунку 3.7 можна побачити карту тестової сцени. На ній позначено початкові точки баз сторін, розміщені на карті дерева, а також сірі зони – недоступні для гравців частини карти.

Така компоновка карти досить типова для стратегій. Гравців розміщують у найдальших точках карти, що для прямокутних або квадратних карт є діагоналлю, даючи простір для розвитку та встановлення зони контролю.

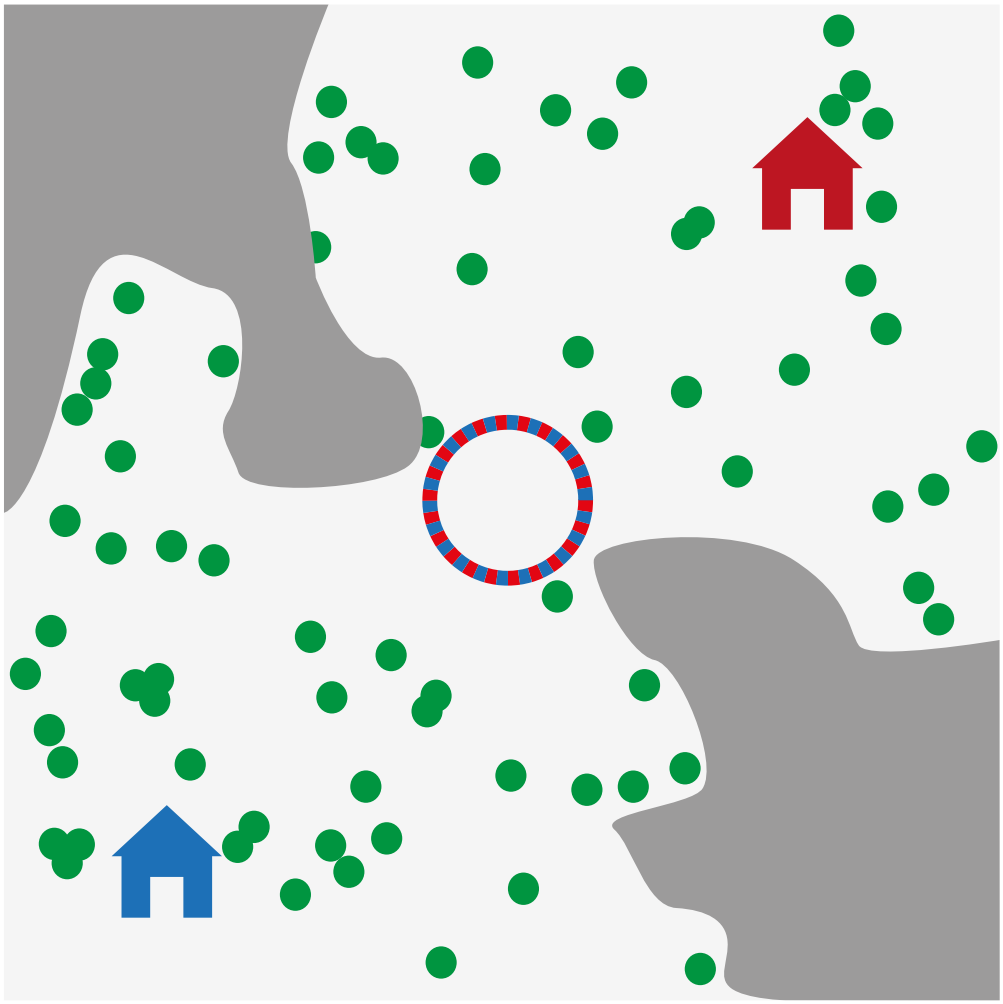

Рисунок 3.7 – Карта тестового середовища

В центрі карти позначено коло, це стратегічна позиція. Оскільки штучний інтелект не може тренувати одразу всіх агентів, потрібно «провокувати» його на активні бойові дії для боротьби з супротивником. Ця карта ідеально виконує дану функцію, оскільки у гравців немає можливості обійти з флангу чи використати більш складну стратегію. Такий формат карти передбачає єдиний шлях до перемоги – прямі зіткнення з бойовими одиницями супротивника в центрі карти. Для перемоги в такому випадку потрібні активний мікроконтроль військ, потужна економіка, здатна створювати війська з такою ж швидкістю, як супротивник. Роль розвідки в умовах даної карти зведена но мінімуму, перевагою гравців буде правильне управління бойовими одиницями та розвиток економіки.

Для збору додаткової інформації щодо перебігу бою в центрі карти встановлено зону, контроль якої надає додаткові бали та ресурси. Збалансованість карти очевидна – дерева не є частиною ресурсної бази, а простір для побудови об’єктів обмежується зоною навколо центральної будівлі.

3.4 Опис основних елементів та механік тестового середовища

Створене тестове середовище є прототипом типової гри в жанрі стратегії в реальному часі з специфікою середньовіччя. Основні механіки гри можна умовно розділити на економічні, бойові та технологічні.

До економічних механік відносяться всі компоненти гри, які впливають на розвиток гравця та кількість ресурсів, які він отримує. Оскільки одна з основних цілей розробки прототипу тестування швидкодії роботи штучного інтелекту та інших елементів гри на мобільних пристроях – при розробці механік гри потрібно це врахувати.

В мобільних іграх завжди існує момент спрощення певної частини функціоналу, це необхідно для створення комфортного ігрового процесу для гравців. Складність управління на телефоні призводить до необхідності скорочення певних механік.

Однією з базових механік стратегії є збір ресурсів та їх балансування. Через обмеженість в управлінні на мобільних пристроях, керування великою кількістю будівель, що добувають ресурси, стає складним завданням. Прибрати механіку видобування ресурсів не можна, адже тоді стратегія в реальному часі перестану бути подібною до інших ігор в жанрі. Єдиним раціональним виходом є варіант спростити систему збору ресурсів.

Скоротивши економічну систему до видобування одного ресурсу та керування показниками заповненості будівель вдасться створити ефективну і просту модель, що може задовольнить користувача, а також збалансувати використання мобільних пристроїв для гри у стратегії.

На відміну від монітора комп’ютера екран планшета чи телефона набагато менший, що призводить до зменшення об’єму інформації, яку користувач може отримати не переміщаючи камеру. Додатковою проблемою є те, що на відміну від інших електронних пристроїв, елемент вводу інформації для планшета і смартфона є водночас екраном. В більшості популярних стратегій створені так звані гарячі клавіші, які дозволяють користувачам швидко викликати потрібну дію без переходу на базу наприклад. Також, важливим елементом інтерфейсу стратегій є можливість ділити війська на окремі групи, для спрощення управління.

В інтерфейсі користувача необхідно врахувати всі ці особливості ігор даного жанру. При проектуванні інтерфейсу користувача було прийнято рішення щодо використання максимальних можливостей для спрощення управління. На наступному рисунку представлено схему розміщення елементів управління для користувача.

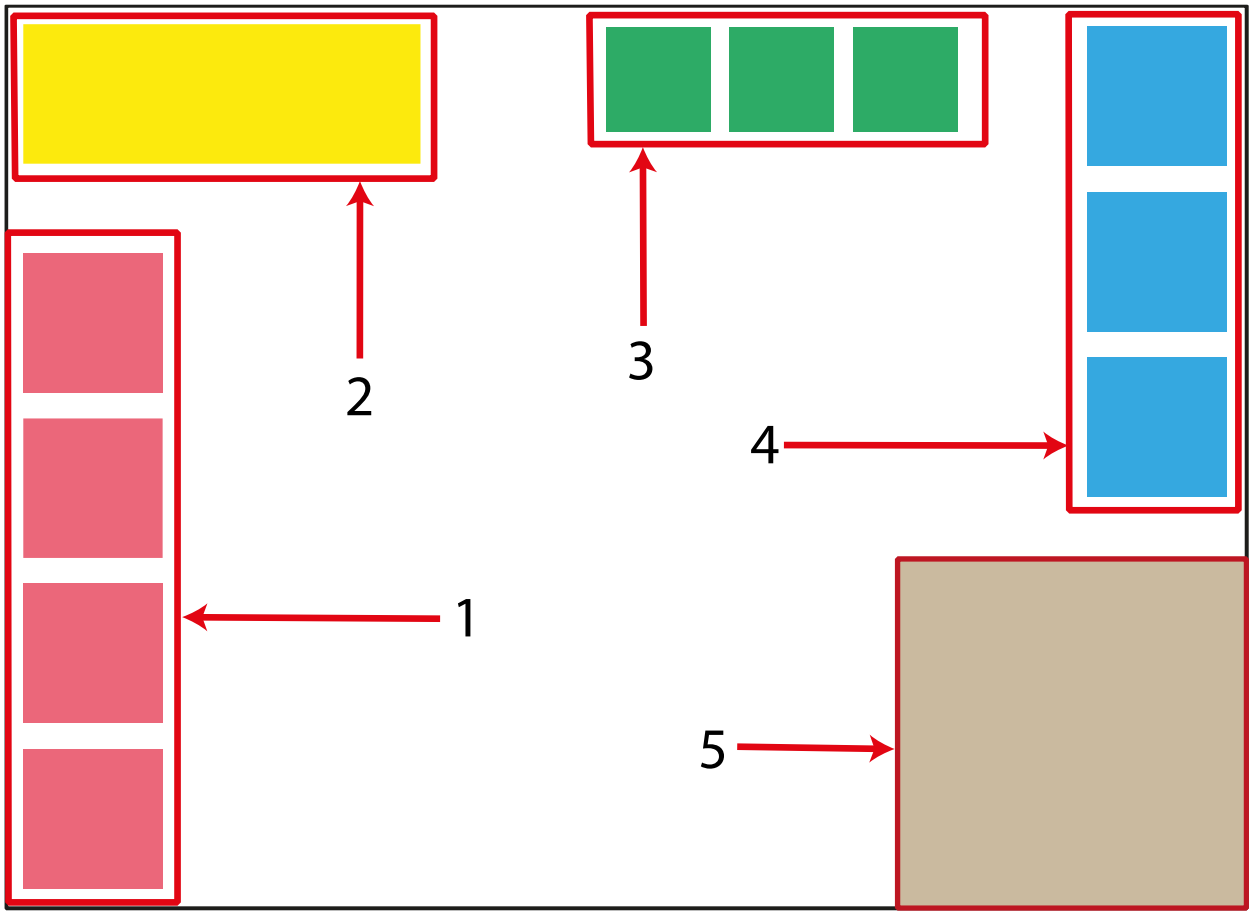

Рисунок 3.8 – Макет інтерфейсу для користувачів мобільних пристроїв

Загалом весь інтерфейс користувача можна розділити на 5 умовних зон. Всі елементи інтерфейсу знаходяться безпосередньо над частиною ігрового поля, на яку наведена камера в даний момент. Взаємодіяти з полем через елементи інтерфейсу неможливо.

Отже, зони управління в інтерфейсі користувача мають наступні призначення:

1. Панель основних доступних дій на елементі (або панель наказів). Ця панель складається з набору кнопок, значення та механізм дії яких залежить від обраного елемента, або його відсутності. Всі основні можливості кожного окремого бійця або будівлі будуть показуватись та будуть доступні до виклику за допомогою цієї панелі

2. Панель даних про поточний стан гри. На цій частині інтерфейсу користувача показується. Такі економічні показники як кількість ресурсів, об’єми їх збільшення, кількість зайнятих будинків та залишок вільного місця для найму військ.

3. Панель даних про поточні ефекти, які впливають на гравця та підконтрольні йому об’єкти на карті. Наприклад, якщо гравець захоплює стратегічну точку – з’являється два ефекти, які будуть негайно приінені до гравця: ефект збільшення видобутку ресурсів та ефект набору додаткових балів перемоги.

4. Панель швидкого доступу. Працює подібно до гарячих клавіш у звичайних комп’ютерних стратегіях, обрану частину військ можна зібрати в групу, а потім за допомогою панелі швидкого доступу мати можливість перейти в їх місце розташування та одразу ж виділити всі бойові одиниці в групі. Така можливість є дуже ефективною при управлінні великими групами бойових одиниць або при боях у великій кількості точок одночасно.

5. Панель міні-карти. Міні-карта використовуються як швидкий та зручний спосіб перейти до потрібного місця на карті, але основний її функціонал – це можливість швидкої реакції на події через спрощену передачі всіх подій на карті.

Панель наказів призначена для управління користувачем окремими бойовими одиницями чи будівлями, а також їх групами. Однак накази в панелі можуть бути дуже різними, і в основному вони залежать від типу об’єкту або одиниці, яка на даний момент вибрана.

У наступній таблиці надано інформацію щодо можливих наказів, які з’являються в панелі. Варто відзначити, що деякі бойові одиниці мають специфічні можливості, які не будуть вказані в таблиці 3.2, але можуть бути викликані гравцем при виборі одиниці відповідного типу.

Таблиця 3.2 – Ігрові об’єкти та доступні гравцям накази

Тип ігрового об’єкта	Назви об’єктів, що входять до обраного типу	Можливі накази для об’єктів
Бойові одиниці	•Легкий піхотинець •Важкий піхотинець	• Рух до точки • Рух до точки з атакою всіх на шляху • Оборона позиції • Патрулювання між двома обраними точками
Робочі одиниці	• Працівник (робітник)	• Рух до точки • Побудова будівлі • Допомога у будівництві • Робота у ресурсній будівлі
Командні будівлі	• Цитадель • Ковальня	• Вивчити технологію • Найняти працівника • Викликати вільних працівників
Ресурсні будівлі	• Лісозаготівельня • Ферма	• Розвинути систему видобутку ресурсів
Військові будівлі	• Казарма • Дім	• Найняти бойову одиницю • Вивчити технологію

Серед рухомих об’єктів гравцю доступно три різновиди одиниць. Це легка та важка піхота, а також працівники. Причина такої обмеженості в кількості типів полягає у необхідності послідовного навчання штучного

інтелекту різним особливостям управління військами. Наприклад, важкі піхотинці можуть ефективно протидіяти скупченням бойових одиниць супротивника різного типу, однак вони набагато повільніші за легку піхоту.

Для можливості подальшого розвитку прототипу та збільшення кількості унікальних бойових одиниць, було обрано моделі, складені з набору елементів, що можна комбінувати у будь якій послідовності.

Кожна модель в грі розділена на 5 елементів:

1. Голова і шолом. Комбінація різних обличь та головних уборів дозволяє створювати унікальні бойові одиниці.

2. Тіло з ногами, а також основна броня. Руки теж відносяться до цієї частини моделі. Не кожна броня візуально добре комбінується з шоломом, також деякі види броні дублюються в «золотому» вигляду – для унікальних бойових одиниць.

3. Ліва рука, а саме – контейнер лівої руки. Різні види щитів та інших захисних елементів. Деяка зброя використовуються двома руками, наприклад, дворучний меч, арбалет або лук. Тому в деяких випадках контейнер лівої руки можна залишити пустим.

4. Права рука, контейнер для зброї та інструментів. Різноманітні види зброї, від списа до молота, можуть бути використані. Для працівників в даному контейнері створені інструменти: молотки, ножі, кирки.

5. Наплічні об’єкти. В поточному стані прототипу дані компоненти не використовуються, однак в подальшому наплічними об’єктами будуть сагайдаки зі стрілами та ящики з ресурсами у працівників.

У прототипі гри на даний момент використано 3 моделі з унікальною комбінацією різних елементів (рисунок).

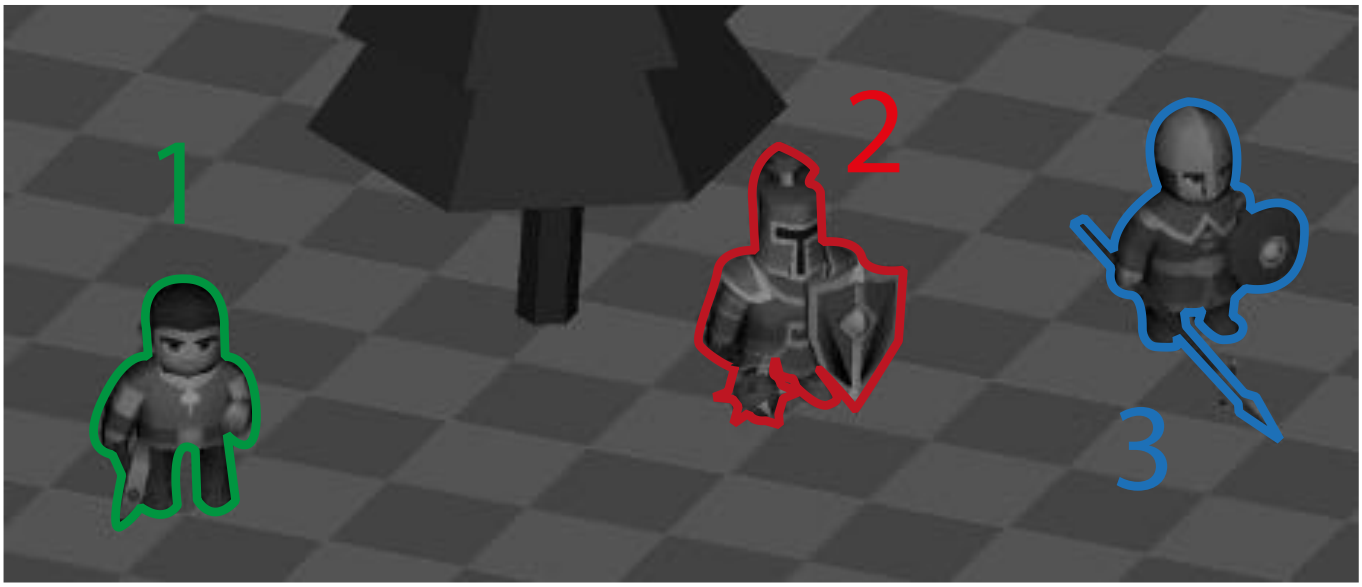

Рисунок 3.9 – Моделі, використані для візуалізації рухомих одиниць в умовах тестового середовища

Під номером 1 на рисунку зображено працівника, в правій руці він тримає кирку, контейнер лівої руки порожній. Елемент голови обраний без будь яких елементів захисту чи шолома, так само й тіло. При нападі працівник використає ніж, але в порівнянні з бойовими одиницями він надто слабкий.

Під номером 2 на рисунку представлена модель важкого піхотинця, він має важку броню, булаву та щит. Самий повільний і потужний бойовий юніт у грі. Для створення цього типу бойових одиниць потрібен певний рівень розвитку технологій – потрібно збудувати Ковальню.

Під номером 3 зображено модель легкого піхотинця. Озброєний списом та легким щитом, він швидший, але слабший за важкого піхотинця, однак доступний раніше і не потребує побудови ковальні для найму.

Кожна з представлених у прототипі одиниць має визначений рівень броні. Деякі бойові одиниці здатні покращити рівень своєї броні після вивчення спеціальних технологій. Для кожної одиниці збережено набір даних у вигляді таблиці, який відображає кількість одиниць пошкодження, які

одиниця здатна нанести тому чи іншому рівню броні в залежності від рівня досвіду.

Досвід отримується під час участі одиницею в бою, кожен піхотинець має певну кількість балів за виконану дію – нанесене пошкодження або знищення бойової одиниці супротивника.

Для прикладу, наведено таку таблицю (таблиця 3.3) для важкого піхотинця (ВП).

Таблиця 3.3 – Кількісна характеристика бойових можливостей ВП

Рівень досвіду / Рівень броні	1	2	3
1	12	15	20
2	10	12	14
3	8	11	13
4	7	10	12

Проаналізувавши дану таблицю можна зробити висновок, що тільки броня вище 2 рівня здатна захистити від удару важкого піхотинця. Інші з першого ж удару отримують критичне ушкодження.

Критичність ушкодження визначається можливістю відновитись після отримання удару. Кожна бойова одиниця та будівля мають декілька рівнів для відновлення здоров’я. Після пробиття одного з рівнів здоров’я без додаткових дій може відновитись лише до останнього «пробитого» рівня. Кожен рівень має розмір 10, і якщо максимальний рівень життів бойової одиниці 50, а поточна кількість 34, то після виходу з бою і 30 секунд спокою життя бойової одиниці почне відновлюватись, однак тільки до рівня 40 одиниць. На рисунку візуалізовано шкалу поділок здоров’я.

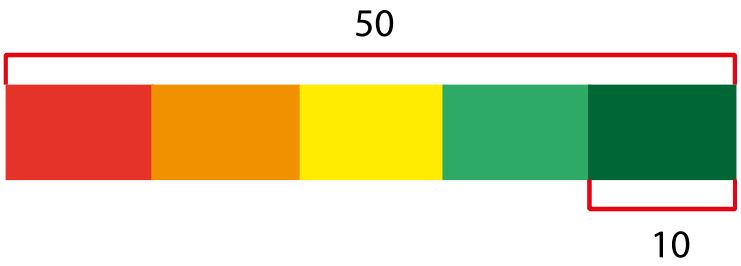

Рисунок 3.10 – Шкала поділок здоров’я для бойових одиниць

Для спрощення користувач буде бачити рівень життів кожної окремої одиниці. Оскільки досить складно одразу аналізувати велику кількість цифр на невеликому екрані, що постійно рухаються разом з бойовими одиницями, для спрощення інтерфейсу над кожною одиницею показник життів буде підсвічуватись тим кольором, який відповідає актуальній кількості одиниць здоров’я.

Для будівель система схожа, однак кожна будівля має в рази більше одиниць здоров’я, тому шкала для будівель збільшена в 4 рази (одна поділка – 40 одиниць здоров’я).

3.5 Висновки до розділу

Цей розділ було присвячено проектуванню середовища тестування, було розглянуто основні готові альтернативні варіанти, через недоліки було прийнято рішення самостійної розробки.

Для розробки було обрано рушій Unity, надано причини такого вибору та порівняння з аналогами.

Описано основні механіки тестової гри, надано макет інтерфейсу та розглянуто деякі моменти, що можуть викликати складнощі при розробці. Окрім цього спроектовано тестову карту та обґрунтовано вибір саме такої структури.

4 ПРОЕКТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ

4.1 Огляд варіантів для створення серверної частини

В сучасних іграх підключення та синхронізація з централізованим сервером вважається стандартним підходом. Для ігрових продуктів найчастіше важливим є конкуренція між гравцями – це спричиняє більшу зацікавленість та готовність змагатись з людьми, адже змагання з штучним інтелектом часто стає нецікавим – супротивник, що завжди діє по одному сценарію навряд чи буде залишатись цікавим після вивчення гравцем стратегії. Тому практично завжди серверна частина в ігрових продуктах має назву мультиплеєр – через можливість взаємодії гравців між собою.

Умовно мультиплеєр, або багатокористувацький режим, в іграх можна розділити на декілька категорій. Перша, і найпростіша з них – таблиці успіху та певні позначки за досягненні. Більшість платформ для продажу ігрових продуктів дозволяють розробникам використовувати власну систему для збереження даних. Наприклад, мобільна платформа PlayMarket дозволяє розробникам використовувати власне API, яке забезпечує повний функціонал і можливості по вищезгаданим функціям. Окремо варто згадати покупки, які можна робити всередині додатків. Цей важливий функціонал найчастіше просто необхідно прив’язувати до платформи, де застосунок буде продаватись і, відповідно, гравці матимуть змогу його завантажити.

Наступною сходинкою розвитку взаємодії ігрових застосунків з сервером є збереження даних гравців, прогресу та інших важливих ігрових елементів. Це дозволяє гравцям покращити процес взаємодії з застосунком, зручно синхронізуватись на інших пристроях, а також відновити прогрес навіть при втраті фізичного пристрою або після видалення застосунку. Важливо, що цей режим не передбачає взаємодії між гравцям в ході ігрової сесії, тільки на рівні синхронізації даних.

І нарешті останньою сходинкою є саме те, що для звичайних гравців означають слова «багатокористувацький режим». Це безпосередня можливість приймати участь в ігровому сеансі одразу декільком гравцям-людям. Згадані вище варіанти між гравцями суттєво обмежують взаємодію, однак вони є більш простими у створенні.

Серверна частина в багатокористувацькому режимі суттєво відрізняється. В залежності від жанру та технічного завдання для продукту розробники обирають найбільш відповідну архітектуру та ставлять акценти на певних моментах.

Специфіка жанру несе в собі різницю для розробників через різні формати даних, які необхідно пересилати іншим гравцям або обробляти якимось чином. Для порівняння можна розглянути чим відрізняється проектування серверів для стратегій, шутерів та ігор в жанрі королівська битва.

Розглянемо жанр королівської битви, який до недавнього часу користувався величезною популярністю. Специфікою цього жанру є велика кількість гравців на одному ігровому полі – від 15 до декількох сотень. Через це технічні рішення повинні враховувати необхідність постійної синхронізації серед величезної кількості комп’ютерів гравців. Інтернет з’єднання та швидкість взаємодії з сервером можуть бути дуже різними. До того ж варто враховувати необхідність перевірок дій користувачів на сервері для виявлення ігрових шахраїв (cheater – чітер), які намагаються порушити правила, встановлені розробниками і отримати незаконну перевагу в ході ігрової сесії.

Для шутерів важливим показником є швидкість синхронізації між гравцями. Найчастіше в сесії приймає участь до 30 гравців, зазвичай команди набагато менші – найпопулярніше режими 4 на 4 або 5 на 5, де сервер взаємодіє з 8 або 10 клієнтами відповідно. Оскільки в іграх цього жанру найчастіше «час для вбивства» (time to kill – TTK) дуже малий, миттєва (з точки зору гравців) синхронізація необхідна для якісної гри та відпрацювання потрапляння в супротивника. Об’єми даних, які мають передаватись, відносно

невеликі, однак швидкість – ключовий показник. Також, у шутерах та сама проблема з шахраями, що і в королівських битвах – це призводить до необхідності співставлення даних, які надходять від клієнтів.

На противагу королівським битвам, у стратегіях у матчі приймає участь найчастіше не більш як 8 гравців, а найбільш поширеними є режими 1 на 1 та 2 на 2. Але складності синхронізації між гравцями знаходяться на зовсім іншому рівні. Якщо в інших жанрах з іншими гравцями, та, відповідно, сервером потрібно синхронізувати тільки персонажа та його одяг/зброю/сумку та її наповнення, в стратегіях між гравцями потрібно синхронізувати сотні, а інколи навіть тисячі об’єктів, які перебувають постійно в стані переміщення, - ігрових юнітів, а також додаткові дані, такі як побудова нової будівлі, створення нових снарядів і напрямку їхнього руху, отримані бали, тощо. Через це при розробці серверів для стратегій основна увага приділяється саме розробці зручного та стислого формату передачі даних між сервером та клієнтами. Окремо варто зауважити про необхідність врахування можливості шахрайських дій в стратегіях. Однією з основних цілей шахраїв є отримання даних з усієї карти, а не тільки з видимої частини. Це дає велику перевагу гравцю в можливості виявлення стратегії та негайної реакції на будь-які зміни. Одним з варіантів вирішення проблеми є надання сервером гравцю тільки тих даних, до яких він має доступ. Однак це призводить до додаткових витратах часу на сервері, а кількість даних є достатньо об’ємною, з часом обробка буде забирати все більше часу.

Для розробки серверної частини в стратегіях немає єдиного «правильного» підходу. Як і у випадку з розробкою штучного інтелекту, в різних компаніях та дослідженнях використовуються власні підходи.

В старих стратегіях, які розроблялись до середини 90-х років часто використовувався підхід прямого з’єднання між клієнтами. Команди, які гравці віддавали своїм підрозділам негайно відправлялись гравцям-супротивникам або союзникам. Проблема такого підходу була очевидною – чим більша кількість гравців, тим складніше обробляти велику кількість

синхронних запитів від великої кількості клієнтів. Поки багатокористувацький режим не став основою ігрових продуктів, пряме з'єднання активно використовувалось і досить ефективно забезпечувало гравцям можливості спільної гри.

Одним з першопроходьців в розробці якісної системи для багатокористувацької гри в стратегії в жанрі реального часу були команда Ensemble Studio, а продуктам, що розроблявся, була серія чи не найвідоміших ігор в жанрі – Епоха Імперій (Age Of Empires). Для першої та другої частин гри використовувались подібні підходи для створення серверів. Вся синхронізація проходила через клієнта-тримача, або хоста, інші клієнти не мали прямого підключення один до одного. Основні завдання на той час перед розробниками поставали досить складні – враховуючи швидкість мережевого з'єднання та потенційні затримки з часом відповіді. Ще однією проблемою була синхронізація з швидкістю відтворення змін на графічному пристрої гравця. Через слабкі процесори і відсутність відеокарти робота з графікою була на низькому рівні, а в іграх режим тривимірності симулювався – насправді все обраховувалось на двовимірній мапі.

Для синхронізації розробники Age Of Empires використали популярний донині підхід з фіксованими точками часу [29]. Суть цього підходу полягала у синхронізації даних між клієнтами тільки в певні рівні моменти часу. Це дозволяло обрати оптимальний час для синхронізації між різними клієнтами. Під час початкової синхронізації кожен з клієнтів розраховував час, необхідний для обміну даними з сервером та внесенням змін в процес графічного відтворення (варто зауважити, що розробники використовували самописний ігровий рушій, в якому процес графічного відтворення (рендерингу) не був прив'язаний до інших подій, на відміну, наприклад, від Unity, де перед кожним кадром та після рендеру викликається набір методів, з яких складається система станів об'єктів у площині їх графічного відтворення). Отримавши початкові дані про стан середовища виконання клієнтів в основній симуляції (на комп'ютері клієнта-тримача)

встановлювався час кожної синхронізації як час найповільнішого клієнта. Після цього всі накази, які гравці віддавали, синхронізувались з основною симуляцією одразу, а з іншими користувачами під час моменту синхронізації в визначений час. Для спрощення позначення розробники використовували назву «крок комунікації» (communication turn), дані з кожного такого кроку починали виконуватись тільки на кроці, умовний індекс якого на 2 більше ніж умовний індекс поточного (рис. 4.1). На це рішення розробників спонукала необхідність синхронізації клієнтів з різними системними можливостями.

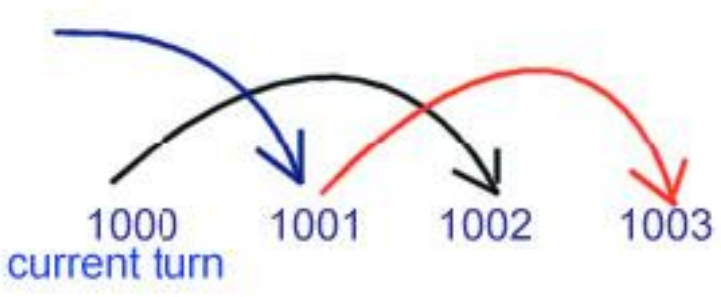

Рисунок 4.1 – Передача наказів через 2 «кроки комунікації»

Однак з розвитком технологій загалом і жанру стратегій зокрема, кількість бойових одиниць під управлінням гравців зростала, багатокористувацькі режими виросли за рамки протистоянь двох гравців, а розвиток мережевих технологій дозволив грати разом людям з різних куточків планети. Через ці суттєві зміни системо прямого з’єднання між клієнтами перестала бути актуальним підходом, на його зміну прийшла клієнт серверна архітектура.

4.2 Вибір архітектури серверної частини

Підхід до архітектури сервера у будь-якому випадку залежить від потреб кінцевих користувачів або технічного завдання (що часто формується на потенційних потребах користувачів). Оскільки застосунки для різних сфер часто мають дуже різну специфіку, єдиного шаблону для створення серверної

частини додатків не існує. Є певний набір підходів та архітектурних шаблонів, які можна використовувати як орієнтири. Для веб-застосунків, які майже завжди потребують певної централізованої серверної частини, основними архітектурними підходами є сервісний і монолітний. Кожен з них має свої переваги та недоліки, які варто врахувати при розробці.

Монолітна архітектура – традиційний підхід до побудови серверної частини веб-застосунків, коли серверна частина складалася з одного великого модуля, що має спільний доступ до ресурсів, баз даних та інших компонентів системи. Всі окремі компоненти системи тісно повязані між собою, і це є як і перевагою, так і недоліком даної архітектури. На рисунку можна побачити приклад монолітної структури серверної частини.

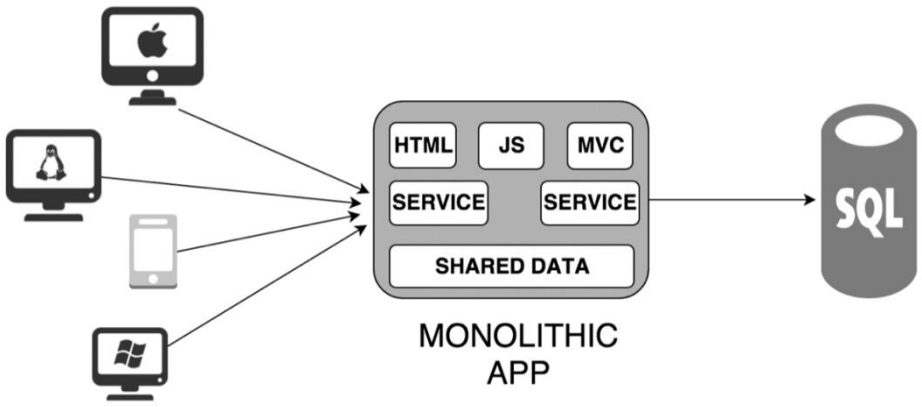

Рисунок 4.2 – Клієнт-серверна монолітна архітектура для веб-застосунків

Серед переваг моноліту можна виділити легке повторне використання у різних частинах додатку різного коду та модулів (наприклад, єдиний модуль для запису та зберігання подій – логування, або надання прав доступу для груп користувачів). Спільне використання ресурсів додатком також можна розглядати як перевага такого архітектурного підходу – це дозволяє відносно легко аналізувати та керувати ресурсами фізичного сервера, які потрібно виділити для стабільної роботи продукту. Також, варто врахувати що

монолітна архітектура дозволяє дуже швидко передавати дані між модулями, оскільки прямий виклик функції (наприклад) завжди буде працювати швидше ніж передача даних до окремого сервісу або стороннього застосунку.

Однак, не дивлячись на вище перелічені переваги, підходи з використанням монолітної архітектури не користуються великою популярністю в наш час, більш того – більшість розробників вважають такі підходи застарілими і не рекомендують використовувати їх в реальних проектах, що розробляються для комерційного використання.

Основними причинами такого ставлення до монолітної архітектури є асоціація з застарілими системами – монолітна архітектура довгий час була традиційною для багатьох продуктів. Через використання єдиного модуля для управління всією серверною частиною з часом виникав ряд проблем, які ускладнювали подальшу підтримку такого продукту. Компоненти системи, які зручно використовувалась в різних модулях, потребували постійної адаптації функціоналу до потреб конкретного модуля. Однак в системах, які знаходяться в стані активного розвитку, кількість модулів постійно збільшується, що призводить до постійного збільшення специфічного для окремих модулів коду в компонентах, що в подальшому призводить до складнощів у використанні (наприклад, методи з подібними назвами та списком параметрів, але різним функціоналом) та підтримці. Окрім цього додавання нових модулів може проходити з використанням нових підходів, що призводить до розділу системи на кілька умовних частин, основною різницею між якими є період розробки та відповідальна за неї частина команди. Крім того, довгострокові проекти на монолітній архітектурі є складними у розуміння для нових розробників, оскільки велика кількість зав’язків між модулями та компонентами не дозволяє з легкістю прочитати код, а головне – зрозуміти його.

З точки зору масштабування, моноліт можливо масштабувати вертикально – збільшувати можливості фізичного сервера, надаючи цим додаткові ресурси для серверної частини. Проблема такого способу

масштабування досить очевидні – можливості серверних машин обмежені, а ціна зростає непропорційно до збільшення кількості оперативної пам’яті та потужності процесора. Горизонтальне масштабування для моноліт також в певній мірі можливе. Серверна частина запускається на декількох окремих (відокремлених) серверних середовищах, що дозволяє масштабувати рівень серверної частини, однак залишаються проблеми з синхронізацією доступу до даних (рівня збереження), а окрім синхронізації проблема виникає через велику кількість паралельних запитів до сховища (бази даних).

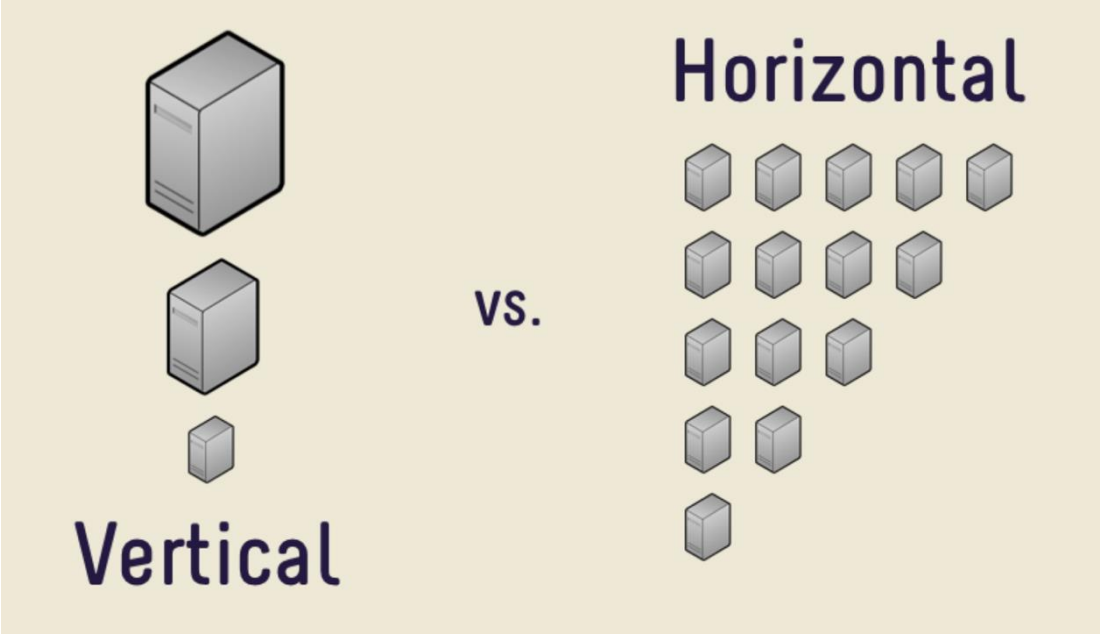

Рисунок 4.3 – Різниця між вертикальним та горизонтальним масштабуванням

Зіткнувшись з цими проблемами моноліту архітектори програмного забезпечення запропонували рішення, що полягає у розділенні монолітної структури на окремі, умовно незалежні частини – сервіси.

Сервісна архітектура – це узагальнена назва методів по розділенню серверної частини на певний набір компонентів, кожен з яких буде відповідати за чітко визначений набір функції.

Першим підходом, що активно використовувався розробниками та архітекторами була сервісно-орієнтована архітектура (service-oriented

architecture, SOA). Такий архітектурний підхід дозволяв чітко відокремити різні компоненти системи, розділивши єдиний сервер на декілька сервісів, і кожен з них має досить широку, але при цьому чітко обмежену, функціональність. В порівнянні з монолітом така система мала багато переваг через відносну гнучкість та чіткі границі між сервісами. Однак на багатьох рівнях певні елемент, наприклад база даних, все ще єдина для всіх сервісів. І в цьому полягає важлива проблема – неможливість масштабування окремих компонентів системи. Оскільки високонавантажені продукти з часом ставали все більш реальною проблемою, рішення в сфері масштабування стали ключовим фактором для вибору архітектури.

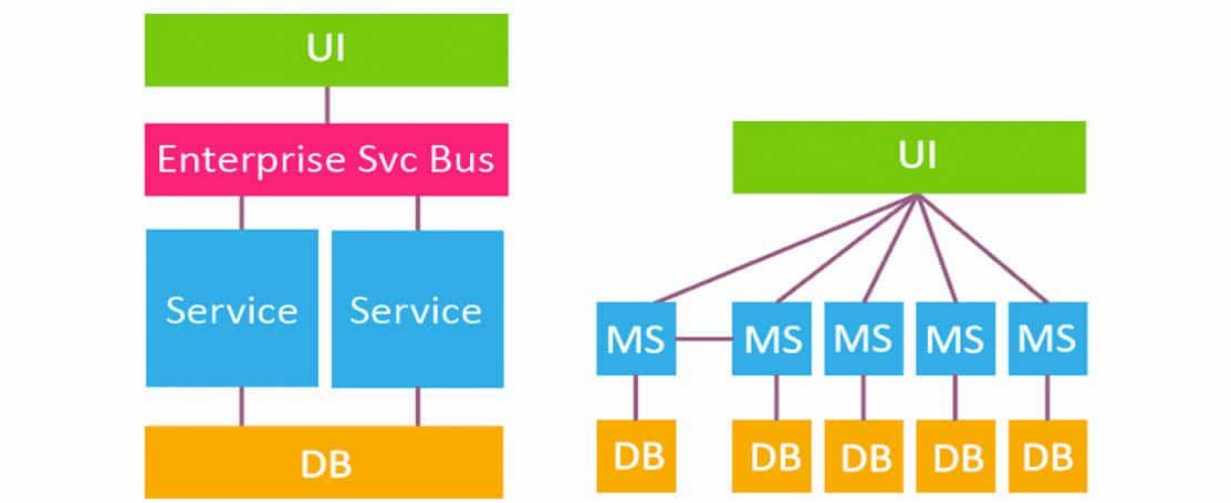

Рисунок 4.4 – Сервісна (зліва) та мікросервісна архітектура (справа)

З розвитком сервісно-орієнтованого підходу розвивалась і підходи самого процесу розділення частин моноліту на сервіси. З часом розмір сервісів зменшувався, основною причиною цього була необхідність активного і, що найголовніше, синхронного розвитку продукту в різних його частинах. Це призвело до появи так званого мікросервісного підходу – архітектури, в якій серверна частина продукту розділяється на набір незалежних компонентів з своїми сховищами даних та системою масштабування. Варто зауважити, що

мікросервісний і сервісно-орієнтовані підходи мають суттєву різницю, яку можна побачити у табличці нижче.

Таблиця 4.1 – Різниця між сервісно-орієнтованим та мікросервісним підходами

	Сервісно-орієнтована	Мікросервісна
Рівень зберігання даних	Модель SOA має єдиний рівень зберігання даних, який поділяють усі сервіси в застосунку.	Додатки мікросервісів в основному присвячують базу даних або інший тип пам'яті службам, які цього потребують.
Основне завдання	Орієнтований на максимальне багаторазове використання сервісів.	Основний акцент на декомпозиції системи та її функціоналу.
Наслідки необхідності розширення	Вимагає модифікації моноліта.	Вимагають створення нового сервісу
Можливості CI/CD	DevOps і безперервна доставка можливі, але не є стандартом	Акцент на DevOps та безперервній доставці
Призначення серверу	Призначений для спільного використання сервісів у різних службах.	Призначений для розміщення сервісів, які можуть функціонувати незалежно.
Рівень доступу	Часто включає спільний доступ до компонентів	Як правило, він не включає спільний доступ до компонентів
Сховища даних	Передбачає обмін сховищем даних між службами	Кожен сервіс може мати незалежне зберігання даних.
Обмін даними	Спілкується через ESB	Обмін даними через шар API
Основна перевага	Покладається на спільне використання ресурсів	Базується на обмеженому контексті зв'язку.
Розгортання	Менша гнучкість у розгортанні	Швидке та просте розгортання.
Розміри технологічного стеку	Технологічний стек SOA нижчий у порівнянні з Мікросервісом.	Стек технологій мікросервісу може бути дуже великим.
Кількість служб, які можуть підтримуватись	Додаток SOA, що складається з двох або трьох служб.	Додаток Microservices може мати десятки послуг.

Продовження таблиця 4.1

Завдання сервісів	Сервіси SOA побудовані для виконання численних бізнес-завдань.	Вони побудовані для виконання одного бізнес-завдання.
Складність розгортання	Розгортання - це трудомісткий процес.	Розгортання є простим і менш трудомістким.
Зв'язок домену і компонентів системи	Компоненти бізнес-логіки зберігаються всередині одного домену служби, простий протокол обміну.	Бізнес-логіка може існувати в корпоративній сервісній шині доменів, як окремі рівні між службами.
Обмін повідомленнями між компонентами	Використовує промислову сервісну шину (ESB) для зв'язку	Використовує менш складну систему обміну повідомленнями
Об’єми програмного забезпечення	Розмір програмного забезпечення більший, ніж будь-яке звичайне програмне забезпечення	Розмір програмного забезпечення невеликий
Акцент	Максимально багаторазове використання	Акцент на декомпозицію.
Можливості масштабування	Менш масштабована архітектура.	Високо масштабована архітектура.

Мікросервісна архітектура в ідеальному варіанті чітко відображає комунікацію між різними відділами організації, для якої створюється продукт. Саме такий варіант архітектури згідно з законом Конвея, є ідеальним для швидкої та якісної розробки.

Окремим пунктом, що додає мікросервісному архітектурному підходу переваг, є можливість постійної доставки оновлень до кожного окремого мікросервісу без зміни інших. Це дозволяє частинам команди працювати максимально незалежно і робити зміни тільки у ту частину продукту, що є їх сферою відповідальності.

При використанні мікросервісів також є можливість масштабувати окремі компоненти (мікросервіси) в системі, при цьому не змінюючи потужності серверів для всіх інших. Це просто неможливо зробити в моноліті,

адже цей підхід використовує єдиний фізичний сервер з його процесорними потужностями та об’ємом оперативної пам’яті.

Важливою функціональною особливістю мікросервісів є повна незалежність та автономність. Це дозволяє досить просто робити декомпозицію поточного мікросервісу на декілька дочірніх, при цьому не змінюючи структуру взаємодії з уже існуючим мікросервісом, а значить розширення системи не потребує фактичних дій від інших команд розробки. Потреба в такому поділі одного мікросервісу на декілька менших може виникнути через занадто велике розширення функціоналу, і виникнення у мікросервісі певних проблем.

Для прикладу можна розглянути розділення мікросервісу повідомлень (Notifications) на невелику сервісну систему. Повідомлення для користувачів є практично в будь-якій системі. Розглянемо мобільний ігровий продукт, типовий представник стратегій. Реалізованими функціями є можливість автономної гри проти ботів, гри з використанням мережі між гравцями, а також функціонал турнірних битв і, відповідно, таблиці успіхів. Серед неігрових можливостей, є функціонал покупок певних елементів гри. Окрім того між гравцями є можливість вести чат, створювати власні матчі та запрошувати як глядачів на поточний матч. Такий функціонал є досить типовим для всіх ігор, не тільки в жанрі стратегій в реальному часі. Частина цього продукту потребує функціоналу повідомлень користувачів: перед турнірним матчем гравцю повинно прийти нагадування, після покупки на електронну пошту користувача має прийти електронний лист – підтвердження покупки та чек. Після отримання повідомлення від іншого користувача на телефон отримувача повинно прийти пуш повідомлення. При наданні користувачем номеру телефона найважливіші повідомлення можна надсилати на номер, також повідомлення для телефону можуть використовуватись як додатковий рівень безпеки.

Якщо для серверної частини використана мікросервісна частина, то з великою вірогідністю для надсилання повідомлень користувачам було

створено окремий сервіс, Notifications Service. За допомогою брокера повідомлень сервіси отримують запити на надсилання різнотипних нагадувань для користувачів від інших мікросервісів системи.

Загальноприйнятим підходом наразі є розділення (або декомпозиція) мікросервісів на максимальну кількість сервісів для спрощеного управління і подальшої розробки. Якщо розглянути мікросервіс повідомлень, то можна розділити його на умовні чотири частини. Обробка вхідних повідомлень і певна додаткова логіка, необхідна перед відправкою, є спільною частиною для ще трьох елементів, кожен з них – відправка кожним з описаних вище способів.

Найчастіше в системах існує чітке розділення засобів надсилання різних типів повідомлень. Для електронної пошти може бути використаний сторонній сервіс на зразок MailGun, для SMS повідомлень існують інші сервіси (наприклад, Twilio), Push-повідомлення обробляються власне додатком на смартфоні або планшеті користувача, однак для їх надсилання потрібен постійно діючий канал зв’язку з використанням для прикладу веб-сокетів. У кожного з цих сервісів є власна документація і API для взаємодії. Єдиний зв'язок між ними – це логічний рівень, який обробляє повідомлення з брокера та викликає функціонал для надсилання. Проаналізувавши ці твердження можна дійти до висновку, що мікросервіс може бути розділений на менші частини, кожна з яких відповідатиме за чіткий обмежений ряд функцій (рисунок 4.5).

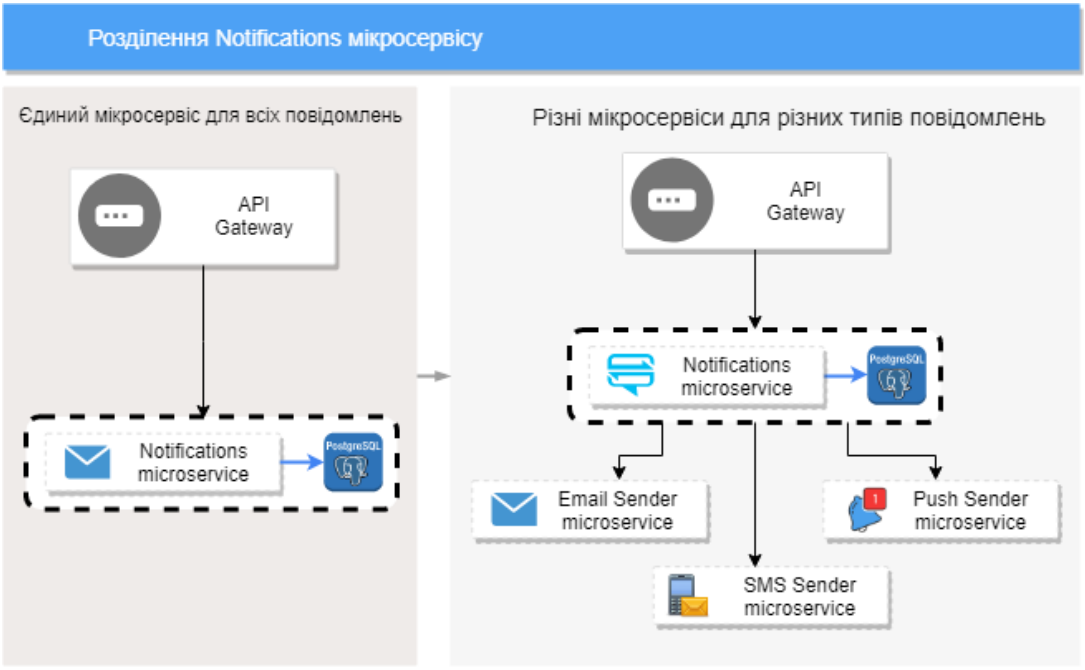

Рисунок 4.5 – Розділення мікросервісу повідомлень на незалежні компоненти

Таке розділення ніяк не впливає на взаємодію з іншими елементами системи та ніяким чином не змінює взаємодію інших мікросервісів з сервісом надсилання повідомлень.

Окрім мікросервісів з розвитком хмарних технологій з’являється все більше можливостей для управління системою з тисячами. Це призвело до появи підходу, що назвали нано-сервісним. Специфікою цього підходу є розділення системи на окремі функції – «нано-сервіси», які працюють за допомогою технологій Azure Functions або AWS Functions. Такий підхід дозволяє керувати ресурсами і масштабувати елементи системи на найнижчому рівні, однак ускладнює розгортання системи та відслідковування проблем.

4.3 Аналіз застосунку та серверної частини

Для вибору архітектури та визначення модулів або сервісів потрібно чітко розуміти особливості системи. Різні сервіси та модулі можуть

потребувати специфічних налаштувань фізичного сервера або конфігурації сховища даних.

Перш за все, необхідно зробити систему передачі досить великих об’ємів даних за короткий час – для багатокористувацького режиму гри. Ключова проблема цієї частини функціоналу – швидкість. Звичайно, в більшій мірі це залежить від користувачів, їх інтернет-провайдера та можливостей телефона чи планшета, однак сервер повинен витримувати високі навантаження і при цьому не зменшувати швидкості обробки та передачі даних. Важливо щоб всі інші активності, які проводяться на стороні сервера не впливали на швидкість обміну даними між гравцями. Конкретно в випадку тестового середовища, яке розробляється, основною задачею цієї частини буде записування всіх даних та короткочасне збереження кожного пакета, щоб в подальшому була можливість точно повторити гру та отримати з неї необхідні дані. Даний функціонал важливий для майбутнього навчання моделі та налізу ситуації. Для гравців важливий елемент перегляду матчів – оскільки під жанр стратегій в реальному часі потрапляють досить складні ігри, часто для розуміння допущених помилок потрібно проаналізувати попередню гру, а функціонал, що зберігає останні ігрові сесії дозволяє це зробити. Увагу треба звернути і на сховище даних – робота з базою часто може сповільнювати роботу серверної частини, тому варто використовувати бази даних, основною перевагою яких є швидкість роботи. Протокол обміну даними з сервером має бути легковісним та швидкісним, важливим елементом даного сервісу є виправлення помилок синхронізації з гравцями.

Наступним компонентом системи є сервіс, що зберігає власне ігрові дані та проводить всі операції взаємодії з сервером ігрового застосунку на кінцевому пристрої користувача, що не стосуються напряму ігрової сесії. Цей сервіс надає дані про ігрову історію кожного окремого користувача і історію результатів боїв з штучним інтелектом. На ньому зберігаються також ключові моменти кожного поєдинку – накази гравців та події, що виникають в ході ігрової сесії. Окрім того кожна бойова одиниця, представлена в грі, всі дані

про чисельні характеристики зберігає на сервері, під час завантаження гри на пристрої користувача дані будуть оновлені. Це дозволяє проводити додаткову перевірку даних та впевнитись у відсутності порушень ігрових правил з сторони гравців. Ще однією важливою функцією є створення матчів та підбір відповідних гравців або рівня штучного інтелекту. Дана частина системи не є критичною з точки зору швидкості, взаємодія між користувачем та сервісом може відбуватись з використанням стандартів HTTP(S). Сховище даних для компонента не є критичною точкою, загалом можна використати реляційну базу даних з чіткою структурою, наприклад PostgreSQL або MSSQL.

Окремо варто виділити компонент взаємодії з користувачем під час авторизації та автентифікації користувача, а також збереження даних про профіль користувача. Критичною точкою цієї частини системи є безпека – дані користувачів не мають потрапити до третіх осіб, тому варто використовувати шифрування частини даних. Це призводить до зменшення швидкості обробки даних та отримання відповіді на запити, але це виправдане рішення, оскільки безпека даних має бути на першому місці. База даних для цих компонентів має бути надійною та мати додаткові засоби для захисту даних.

Наступним компонентом системи є сервіс, що буде відповідати за навчання штучного інтелекту та зберігатиме всі дані про моделі. Ключова проблема даного компоненту систему – необхідність використання великої кількості процесорного часу для обробки даних, навчання моделі, аналізу подій та наказів в обраній ігровій сесії для створення алгоритму дій на основі прецедентів. У зв’язку з цим є велика вірогідність того, що в подальшому потрібно буде масштабувати цей сервіс. Також, для навчання моделі в майбутньому потрібна буде система серверної симуляції. Це дозволить аналізувати кожну ігрову ситуацію при різних варіантах комбінації рішень, розраховувати найбільш успішний варіант та навчати штучні нейронні мережі згідно такого варіанту. Оскільки для симуляції необхідно обрахувати велику кількість можливих комбінацій наказів та взаємодій бойових одиниць,

симуляції потребуватимуть великої потужності процесора та об’ємів оперативної пам’яті.

Проаналізувавши потреби системи та специфіку певних її компонентів, можна одразу зробити висновок, що єдиний сервер – монолітна архітектура – не підходить через складність управління певними частинами ресурсів та проблеми з масштабуванням. Підхід з використанням сервісів є оптимальним, оскільки в системі існує набір відокремлених компонентів, кожен з яких потребує специфічної роботи. Однак видів архітектури, що в своїй основі містить сервіси, велика кількість. Класичний сервісно-орієнтований підхід не підходить для ігрового додатку, оскільки використовує спільне сховище даних. Різні компоненти розроблюваної системи потребують власної реалізації сховища і що головне – потребують різних налаштувань бази. Тому оптимальним можна вважати варіант мікросервісної архітектури, в якій різні частини системи будуть повністю незалежними. Архітектуру системи з використанням мікросервісів можна побачити на рисунку 4.6.

Важливим елементом системи є комунікація між сервісами та передача даних для синхронізації. Такий функціонал можна реалізувати за допомогою прямих запитів між мікросервісами, однак проблема може виникнути при спробі балансування навантаження на сервіс, адже кожен запит під час обробки даних від користувача або синхронізації виконується одразу.

Рішенням даної проблеми є використання брокерів повідомлень, що дозволяє зберігати запити до сервісу та надає можливість серверу в порядку черги оброблювати дані.

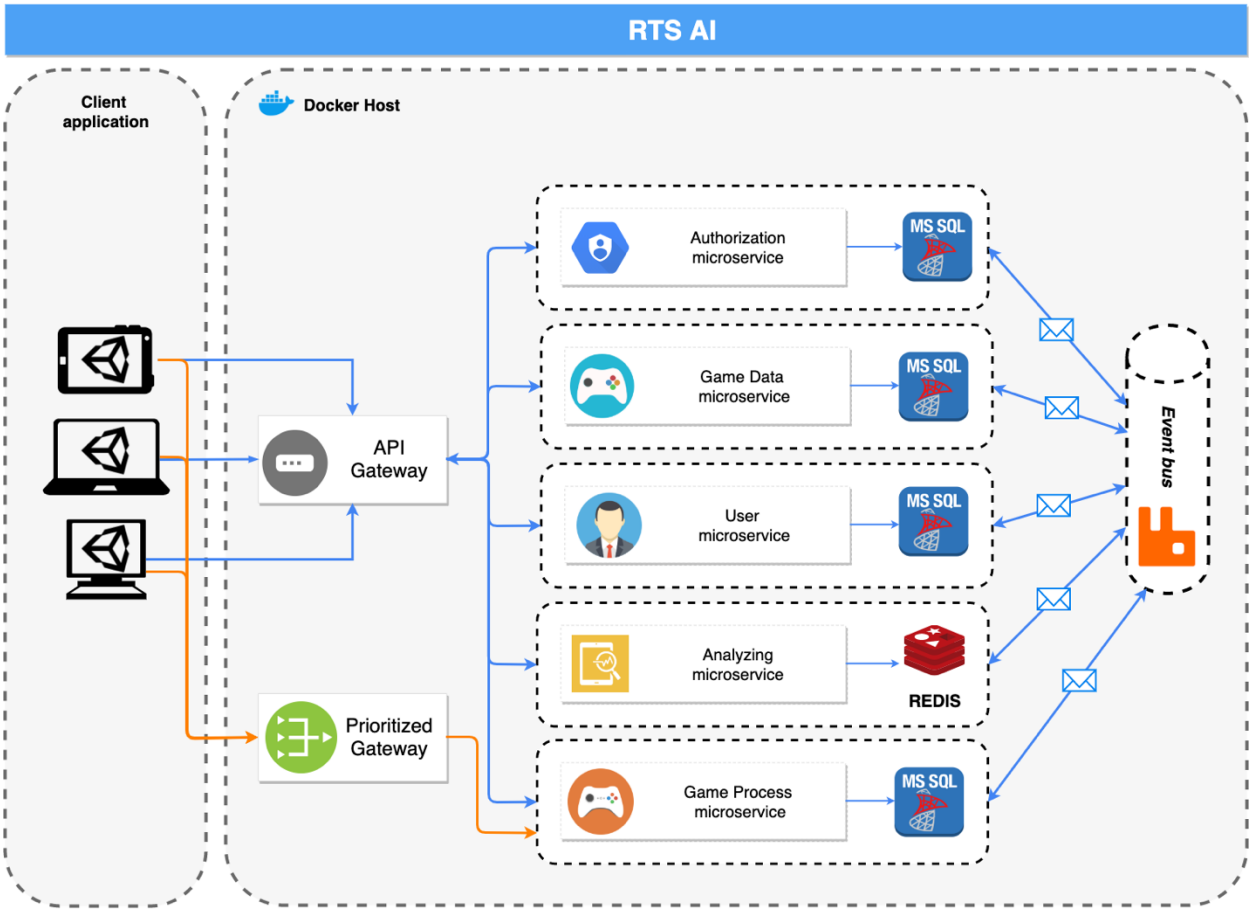

Рисунок 4.6 – Архітектура системи

RabbitMQ є однією з багатьох технологій, що реалізує можливості брокера повідомлень. Серед можливостей даної технології варто відмітити асинхронну обробку запитів та можливість паралельної обробки одразу декількох партій даних. Для зручності використання при розробці створено драйвери для найбільш популярних мов програмування: Java, .NET, PHP, Python, JavaScript, Ruby, Go, та інші, що дозволяє використовати різні технології в сервісах, при цьому користуватись єдиними брокером повідомлень.

При необхідності RabbitMQ має можливості для масштабування горизонтально. Також створено API для самостійного розширення функціоналу або проведення глибокого налаштування системи [30].

Обмін проходить з використанням різних протоколів в залежності від потреб користувача. Протоколи для обміну, які використовуються в RabbitMQ:

− AMQP 0-9-1;

− STOMP;

− MQTT;

− AMQP 1.0;

− HTTP;

− WebSockets.

4.4 Компонент первинної обробки запитів користувачів.

Для взаємодії користувачів (в випадку системи для ігрового застосунку: взаємодії застосунку на кінцевому пристрої користувача з серверною частиною) з сервером існують 2 основні підходи:

- Пряме з’єднання клієнтського додатку з серверною частиною;
- Компонент-посередник між клієнтським додатком та серверною частиною.

Перший полягає використання прямого доступу до мікросервісів, коли запити від користувача на сервер ідуть безпосередньо до потрібних компонентів. Проблема такого підходу очевидна – система перестає бути гнучкою, а робота з кодом, який виконується на стороні користувача, ускладняються, оскільки взаємодіяти одразу з великою кількістю сервісів по схемі побудови логіки нагадує роботу з великою кількістю серверів. До того ж, серверна частина стає залежною від користувача через прямі зв’язки у клієнтському додатку.

Окремою проблемою є використання прямого доступу до сервісів та пряму обробку запитів. Для безпеки всією системи це є критичним моментом, оскільки за збереження даних та захист від дій потенційних злочинців буде відповідати командна, які розробляє мікросервіс. Навантаження на систему

також зросте, оскільки при обробці комплексного запиту може бути проведена взаємодія з великою кількістю мікросервісів, кожен з яких проводить додаткові перевірки для захисту інформації.

Другий спосіб, який полягає у використання компонента посередника між користувачами та мікросервісами. Варто зауважити що в системі може бути використано декілька компонентів-посередників, однак всі вони об’єднані однією специфікою в роботі з клієнтами – вони не зберігають даних і всього лише передають запити від клієнтської частини до мікросервісної.

В них може зберігатись певна частини логіки, наприклад, послідовність взаємодій з мікросервісами для обробки запитів та перевірка користувача на предмет безпеки.

На рисунку зображено приклад використання такого компоненту. В рамках розробки системи для розділення запитів під час ігрової сесії та запитів, які не потребують пріоритетної обробки, було створено два компонента – обробника запитів користувачів.

Один направлений на роботу з звичайними запитам – вхід до системи, завантаження ігрових даних, пошук матчу користувачем та інше. По суті це типовий компонент API Gateway для мікросервісної системи, такі використовуються на більшості проектів.

Другий компонент – пріоритетний обробник запитів (Prioritized Gateway), використовується для обробки внутрішньо ігрових процесів, тобто запитів користувачів, які напряму стосуються процесу гри. Також в цьому компоненті є можливість постійного зв’язку з користувачем за допомогою бібліотеки SignalR.

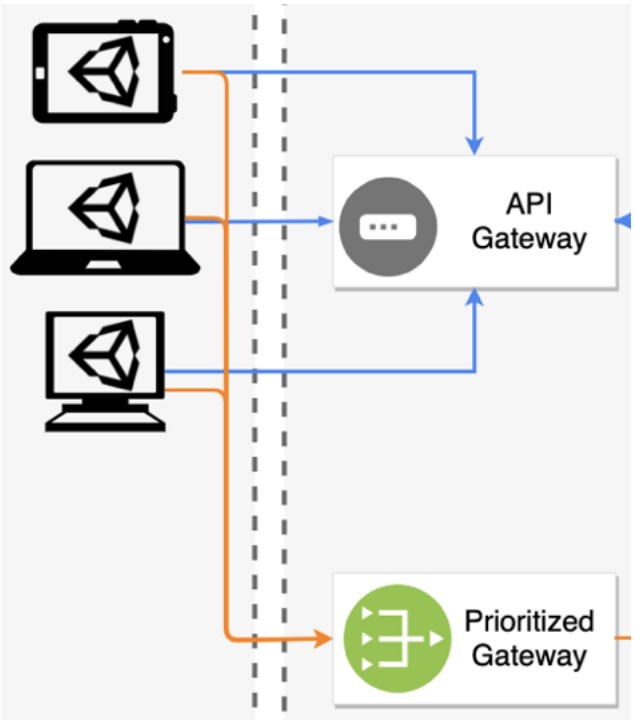

Рисунок 4.7 – Схема розподілу Gateway для обробки звичайних та пріоритетних запитів

SignalR – це бібліотека від Microsoft, що використовується для створення з’єднання в реальному часі між різними компонентами системи. Бібліотека підтримає двостороннє з’єднання для обміну інформацією між клієнтом і сервером.

Вибір саме цієї бібліотеки спричинений її популярністю та легкою інтеграцією в середовище тестування – SignalR сумісний з NetStandart 2.0, а ігровий рушій Unity використовує саме цю версію для роботи з скриптовими файлами.

Важливою перевагою цієї бібліотеки є використання складного механізму клієнт серверного зв’язку з використанням цілого набору протоколів. На даний момент SignalR використовує три протоколи:

- WebSockets;
- Server Side Events (SSE);
- Long Polling.

В залежності від того, які протоколи підтримує клієнт та сервер, бібліотека намагається обрати та використовувати оптимальний. За замовчуванням SignalR використовує WebSockets як найпріоритетніший варіант, якщо цей механізм з якоїсь причини не доступний – починає використовуватись Server Side Events, і в останню чергу переходить до Long Polling.

4.5 Мікросервіси ігрових даних та сесій

До цієї частини системи відносяться мікросервіси, що відповідають за процес ігрової сесії, передача поточного стану від кожного з гравців під час багатокористувацького матчу, підбір супротивників для створення матчів, збереження даних про гру, збереження ігрової інформації та даних (характеристик) про кожну окрему бойову одиницю.

За весь цей функціонал відповідають два мікросервіси – ігрового процесу (GameProcessService) та ігрових даних (GameDataService). Розділення необхідне через специфіку роботу компонента – підбір гравців для матчів та надсилання оброблених великих наборів даних потребує навантаження на сервер. При цьому під час ігрової сесії важливо пересилати дані в найкоротший строк, окрім того дані потрібно частково перевірити, що додає навантаження на сервер.

Мікросервіс ігрового процесу відповідає за передачу даних до користувачів. Серед компонентів первинної обробки є пріоритетний компонент взаємодії з системою – Prioritized Gateway. Варто зауважити що даний компонент не є повноцінним мікросервісом, а існує лише як проміжний елемент для приймання та передачі даних. Через цей gateway проходять дані, які обробляються у процесі гри. Особливістю взаємодії є постійне з’єднання між сервером та користувачем, оскільки важлива висока швидкість, тому для зв’язку з користувачами використовується не тільки підхід «запит-відповідь», а й постійного з’єднання.

В рамках GameProcessService ведуться обрахунки поточного стану гри, саме в ньому реалізована система з фіксованими точками в часі, в яких відбувається синхронізація. Така система дозволяє балансувати навантаження на клієнтських пристроях і на сервері. Всі дані, які готує до обробки сервер зберігаються у тимчасовому сховищі, це потрібно для доступу до повторів гри, а також для можливості завантажити гру в подальшому.

Збереження даних відбувається у базі даних MongoDB. Вибір нереляційної бази даних в цьому випадку цілком виправданий – обмін даними буде виконуватись з використанням протоколу JSON, дані в наборах можуть суттєво відрізнятись в залежності від версії гри

Сама система фіксованої синхронізації працює наступним чином:

1. В ході ігрової сесії користувачі надсилають на сервер всі дії, які вони виконують. Для цього використовується класифікатор дій, який було створено для тестового середовища. Всі ці дії надсилаються до мікросервісу ігрових даних і зберігаються.

2. Обмін даними про стан ігрового середовища відбувається по чітко визначеним періодам часу. В тестовому середовищі використовується єдиний контролер для управління практично усіма можливими діями та об’єктами. Цей контролер має доступ до всіх об’єктів під управлінням гравця. Саме через нього проходять основні дані про ігрову сесію. За один «хід» ігрового процесу обміну даними застосунок отримує оновлені дані, застосовує їх до видимих об’єктів,

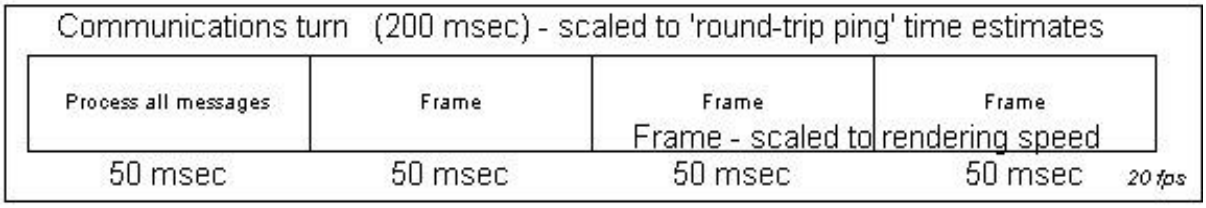

Рисунок 4.8 – Схема взаємодії з серверною частиною з урахуванням затримок різни користувачів

3. Наступним кроком застосунок на стороні клієнта створює власний пакет даних з інформацією про стан бойових одиниць та будівель, які знаходяться під контролем, а також бойові одиниці, які видимі користувачу. Цей пакет даних необхідний для додаткової синхронізації – оскільки кожен клієнт самостійно займається обробкою команд і гра має бути синхронною для всіх користувачів, для того щоб впевнитись в правильності і синхронності клієнтів дані оновляються.

4. Отримавши дані, сервер порівнює дані від різних гравців на предмет розсинхронізації. Перевірка шукає проблеми в розміщенні та інших характеристиках присутніх на полі об’єктів, при знаходженні суттєвих розбіжностей – разом з новим набором команд надсилає гравцю «синхронізаційний» пакет, для переміщення об’єктів в потрібну точку та збільшення/зменшення окремих характеристик.

Така системи взаємодії з сервером дозволяє досить просто перевіряти актуальність даних та порівнювати розбіжності, при виникненні проблем одразу ж реагувати на потенційну розсинхронізацію та виправити ситуацію.

Окрім прийому-передачі даних та команд від користувачів сервіс також відповідає за збереження ігрових команд у сервісі ігрових даних GameDataService. Для цього при появі нової команди від гравця мікросервіс GameProcessService за допомогою RabitMQ направляє повідомлення в GameDataService для внесення в базу даних про новий наказ гравця. Окрім наказів для додаткової синхронізації в базу вносяться події, які відбуваються у грі. В подальшому при необхідності будь яку гру можна буде відтворити на відповідній версії клієнтського застосунку. В деяких моментах відтворення може бути не точним – адже в GameDataService не зберігаються дані про точне положення кожного об’єкта.

GameDataService також відповідає за зберігання наступних даних щодо процесу гри:

- Ігрові сесії та їх поточний стан (наприклад, підготовка, помилка синхронізації, успішно завершено);

- Детальна інформація про учасників ігрової сесії – посилання на гравця, номер команди, місце в рейтингу на момент початку матчу, для бота – варіант штучного інтелекту а також його версія (для кожної частини штучного інтелекту окремо);
- Дані про всі бойові одиниці, будівлі та технології, які є доступними в грі, їх ціну та набір характеристик;
- Таблиці інформації щодо нанесення пошкоджень для різних типів військ та відповідно різного типу броні.

Це спрощений перелік того, що зберігає в собі мікросервіс GameDataService, він є основним сховищем даних про гру.

Окремо варто звернути увагу на збереження в сервісі ігрових даних характеристик типів бойових одиниць та інших об’єктів, з якими можна взаємодіяти в ході ігрової сесії. Для будь якої гри в жанрі стратегії в реальному часі один з ключових показників в хорошого ігрового продукту є продумана та збалансована система фракцій або рас. Це дозволяє урізноманітнити ігровий процес для гравців, та покращити візуальну складову продукту, оскільки між собою будуть боротися бойові одиниці, що відрізняються не лише кольором умовних обладунків, а й самими моделями (у випадку тривимірної графіки) або спрайтами (у випадку двовимірної графіки).

Збереження характеристик на сервері дозволяє змінювати характеристики бійців безпосередньо в базі, при цьому без потреби змінювати код на кінцевому пристрої користувача. Така система в подальшому дозволить вносити зміни в баланс фракцій, а також додаткова можливість перевірити дані, що приходять від користувачів на предмет змін у характеристиках бойових одиниць.

Для реалізації даних мікросервісів було обрано технологію .NET Core. Цей фреймворк зарекомендував себе з найкращої сторони при побудові структурно складних, однак при цьому достатньо швидких серверних частин для веб-застосунків, мобільних додатків та ігор. Додатковою перевагою є простота інтеграції з Unity частин структури сервера, оскільки в обох

технологіях використовується одна й та ж мова програмування, а також обидві технології сумісні з NetStandart, який потрібен для використання бібліотеки SignalR.

4.6 Мікросервіс аналізу та штучного інтелекту

Окрім сервісу, для збереження інформації щодо ігор та сервісу аналізу, необхідно додати сервіс для авторизації користувачів ти сервіс збереження даних гравців. Це дозволить зберігати специфічні для окремих гравців налаштування моделі, що зробить штучний інтелект гнучким та орієнтованим на користувача.

Особливо корисно це вплине на визначення початкових етапів розвитку в межах стратегічних рішень. Різним гравцям подобаються різні підходи для гри, а початковий етап майже завжди задає темп. Агресивні дії зазвичай задають швидкий прийом рішень та постійні сутички з бойовими одиницями супротивника. Для того, щоб штучний інтелект був потужним, але водночас і цікавим, мікросервіс, що відповідає за штучний інтелект, буде зберігати дані кожного гравців, і кожен з них матиме можливість грати на зручному рівні.

Окрім залежності початкової фази від вибору гравців, рівень управління військами в рамках тактичного управління також буде залежати від умінь гравця. З кожною ітерацією навчання нейронна мережа буде ставати все потужнішою і ефективнішою, що зробить її важким супротивником для гравців. Однак сервіс ігрових даних буде відповідати за зберігання коефіцієнтів мережі після кожного з навчань. За допомогою цього функціоналу гравець зможе при бажанні змагатись з менш навченим ботом, ніж максимальний поточний рівень мережі. Для гравців з більшим потенціалом та можливостями будуть доступні боти з вищими показниками. Таким чином у рамках гри, яку можна створити на базі тестового середовища, для гравців усіх рівнів буде доступний зручний тактичний бот, який дозволить комфортно змагатись з ним.

Для ієрархічної мережі задач, яка буде визначати розвиток стратегії бота після початкової фази гри, основними моментами, що впливають на якість та правильність прийняття рішень, є показники пріоритетності та ефективності кожної окремої вітки задач.

Для впливу на цю частину штучного інтелекту також можна використати систему коефіцієнтів для пріоритетності елементів мережі. Це дозволить зменшити ефективність штучного інтелекту, оскільки більш стара версія (номер ітерації навчання менший, за актуальну версію) коефіцієнтів з меншою ефективністю використовує ресурси та реагує на загрози.

Окрім переваг у можливості гнучкого використання штучного інтелекту проти гравців, мікросервіс аналізу також відповідає за навчання моделей для нейронних мереж та зміни в коефіцієнтах ієрархічної мережі.

Саме такий підхід був обраний з декількох причин:

1. Все навантаження через навчання моделі та аналіз відомих рішень переводиться з кінцевих пристроїв користувачів на сервер, що дозволяє зменшити використання процесору грою та надає гравцям з пристроями, що мають низькі показники ефективності, грати без проблем та «зависань» застосунку через перевантаженість компонентів системи;
2. Для навчання моделі використовуються ігрові дані з усіх сесій, які були проведені і синхронізовані з сервером. Це дозволяє вибудувати оптимальний з точки зору ефективності штучний інтелект. Також важливим аспектом є специфіка стратегічних ігор, оскільки в стратегіях гравці часто дотримуються певного напрямку. Навчена система для боротьби з одним «типом» гравців буде повністю неефективною проти іншого «типу», навчання ж на всіх доступних матеріалах дозволяє системі бути гнучкою.
3. Закритість та можливість додаткових методів навчання. Ця важлива перевага стає доступною через використання серверної частини та симуляцій. За допомогою симуляції можна визначити ефективність того чи іншого вибору і розрахувати наскільки правильними є поточні налаштування. Окрім того є можливість створення веб-сервісу для навчання моделі.

4.7 Мікросервіси користувачів і авторизації.

Дан сервіси є критично важливими для роботи системи, оскільки в них зберігається основна частина даних користувачів, які повинні залишитись конфіденційними.

Мікросервіс авторизації відповідає за безпеку даних користувачів та доступ користувачів тільки до певної частини системи, яку надає рівень доступу – роль. Для авторизації в застосунку можна використовувати наступні способи:

- Ввід електронної пошти та пароля;
- Ввід спеціального імені користувача та пароля;
- Використання власного облікового запису в Google;
- Використання власного облікового запису в Facebook;
- Авторизація з використання Apple ID + Game Center або Google Play Games.

AuthService відповідає за функціональну частину та взаємодію з API систем авторизації з використанням облікових записів. Варто зауважити, що останній варіант авторизації відбувається на кінцевому пристрої користувача з встановленим застосунком, тому на сервері зберігається мінімальна кількість інформації щодо користувача.

Окрім функцій авторизації, мікросервіс відповідає і за аутентифікацію користувача, яка відбувається практично з кожним запитом до системи. Такий функціонал на сервісі потребує налаштування фізичної частини та потенційну можливість масштабування даного мікросервісу для оперативної обробки запитів користувачів.

Мікросервіс користувачів (UserService) зберігає дані користувачів, а також конфіденційну інформацію, яка повинна зберігатись у зашифрованому вигляді. Окрім електронної пошти, паролів та телефонів, в даному сервісі буде

зберігатись інформація щодо дій та цінності користувача з точки зору маркетингового просування продукту.

Реферальна система у грі дозволить гравцям отримувати бонуси за нових активних користувачів, які скористались запрошенням гравця. Всі зв’язки між користувачами, а також взаємодія в рамках ігрового продукту (переписка) буде зберігатись в даному сервісі.

Окремо варто виділити функціонал бану користувачів. Оскільки багатокористувацькі ігри часто потрапляють до кола інтересів осіб, які хочуть отримати перевагу в ході ігрової сесії за рахунок заборонених способів, функціонал заборони входу в обліковий запис є необхідним. Заборона може бути обмеженою в часі, або бути постійною.

4.8 Висновки до розділу

В розділі було розглянуто можливості для проектування серверної частини системи. Було обрано мікросервісну архітектуру для системи, надано порівняльну характеристику з іншими архітектурами, обґрунтовано вибір.

Виділено основні мікросервіси, серед яких мікросервіси ігрових даних та ігрового процесу, авторизації та користувацьких даних, а також сервіс навчання штучного інтелекту.

5 РОЗРОБКА ТА ТЕСТУВАННЯ СИСТЕМИ

5.1 Розробка тактичного штучного інтелекту

Для розробки штучного інтелекту було прийнято використовувати платформу .NET. Це спричинено тим, що нейронну мережу та інші елементи штучного інтелекту потрібно вбудувати в користувацький додаток, це дозволить в подальшому гравцям користуватись можливостями гри з ботом навіть за відсутності інтернету, а також прискорить швидкодію і стабільність, оскільки всі обрахунки і рішення будуть прийматись одразу в програмному коді, доступному користувачеві.

Попередня обробка даних потребує певного набору дій, які проводяться поза межами нейронної мережі. В рамках тестового середовища, як показано в структурній схему у додатку Є, створено окремий модуль що відповідає за штучний інтелект. Окрім структури нейронної мережі та завантажених коефіцієнтів ваги та зміщення, модуль реалізує функціонал попередньої обробки.

Наступні методи є основними в попередній обробці даних:

- GetAnalyzingTerritory()
- SplitTerritory(Territory ter)
- HandleZones(List<Zone> zns)
- GetUnitsPower(Zone zn)
- IsOwnUnitsExists(Zone zn)
- CountPower(Zone zn)
- CountInfluence(List<Zone> zns)

Методи працюють послідовно, спочатку визначаючи територію для аналізу, потім розділяючи цю територію на частини, названі зонами.

Після розділу на зони кожна з них проходить обробку – алгоритм визначає кількість військ та загальну «силу» підрозділу в зоні, потім це значення використовується в визначенні впливу.

Для зон, в яких відсутні бойові одиниці подальший аналіз проводитись не буде. За загальну територію в рамках системи класів відповідає клас Territory, кожна зона є об’єктом класу Zone.

Після проходження першого рівня обробки даних для певного списку зон буде застосовано нейронну мережу. Через необхідність використання мережі як у серверній частині так і в рамках користувацького застосунку, Було прийнято рішення створення власного рішення на основі мови C#.

Такий підхід довелось застосувати через ряд проблемних моментів в рушії Unity, що не дозволяють зручно додати пакети з Nuget.

Nuget – це основний пакетний менеджер для проектів, написаних з використанням платформи .NET і є стандартом для розробки. Окрім завантаження пакетів з спеціальних репозиторії в є можливість прямого додавання бібліотек класів у вихідний код Unity проекту.

Це несе в собі певні проблеми, оскільки підхід ручного додавання пакетів надто негнучкий, що призводить до збоїв синхронізації та помилок компіляції.

Використання бібліотеки SugnalR, наприклад, потребує додавання більше як 15 файлів різноманітних бібліотек, що призводить до майбутніх потенційних проблем при збільшенні поточної версії бібліотеки (оскільки бібліотеки при переході на нову версію часто підвищують необхідний рівень версій залежностей – dependencies).

Для створення нейронної мережі основними елементами є нейрони. В обраній схемі класів Neuron відповідає за даний елемент мережі, він реалізує інтерфейс INeuron. Кожен з нейронів розроблюваної мережі належить класу NeuronLayer, що відповідає за кожен окремий шар мережі, незалежно від типу шару та його характеристик.

Кожен з шарів нейронів в свою чергу належить класу NeuralNetwork, що є основним класом для нейронної мережі.

На рисунку 5.1 можна побачити класову структуру.

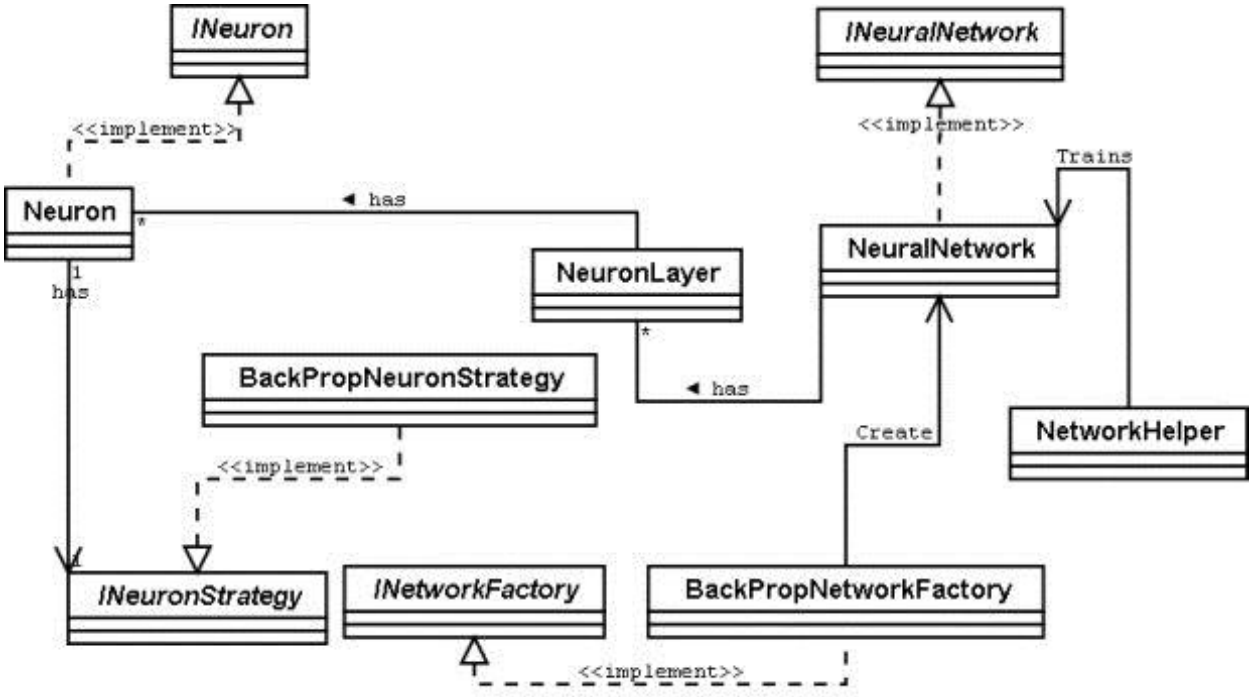

Рисунок 5.1 – Структура класів для розробки нейронної мережі

Оскільки структура нейронної мережі може змінюватись, а коефіцієнти на рівні клієнта будуть залежати від сервера, було прийнято рішення виділити окремий клас BackPropNetworkFactory. Така назва класу спричинена тим, що в рамках системи існують 2 мережеві підходи, для різних елементів системи. Оскільки створювана мережа є нейронною, для її навчання використовується алгоритм звортного поширення (backward propagation).

Даний клас відповідатиме за створення нейронної мережі з потрібними коефіцієнтами після їх завантаження на початковому етапу гри. NetworkHelper, що відповідає за навчання нейронної мережі, має свою реалізацію на серверній частині і по суті є інтерфейсом для надання інформації щодо навчання нейронної мережі.

Як видно із функціональної схеми навчання мережі (додаток Ж), мережу можна навчати двома способами – автоматичної симуляції та коригування наставником.

Спосіб автоматичної симуляції це складний для сервера, однак ефективний спосіб визначення моментальної винагороди за те чи інше рішення в рамках вибору дій для групи.

Враховуючи що вихідний шар нейронної мережі складається з 3 нейронів, що повертають значення в границях від -1 до 1, два з яких задають напрям і один задає режим переходу, можемо визначити що для однієї групи (тобто для масиву бойових одиниць в рамках однієї зони) можна визначити 27 варіантів рішення.

Оскільки всього територія поділяється на 25 квадратів, кількість необхідних симуляцій зростає пропорційно росту кількості зайнятих бойовими одиницями зон.

Така кількість необхідних симуляцій навантажує сервер та заважає ефективно використовувати можливість симуляції для повноцінного навчання моделі. Тому навчання в симуляціях проводиться за спрощеними режимами зонування для прискорення процесу.

Для навчання системи користувачем використовується веб інтерфейс, що дозволяє розміщувати об’єкти на території, передавати для аналізу, переглядати поетапно результати дії алгоритму обробки та коригувати рішення алгоритму. Такий спосіб навчання нейромережі набагато швидший, оскільки правильний варіант подається готовим і не потребує додаткових затрат ресурсів. Проблемою такого підходу є негарантований правильний результат від людини.

Саме навчання мереже – по суті зміна коефіцієнтів, відбувається за типовими алгоритмами. На відміну від людського сприйняття для комп’ютера нейронні мережі по суті представлені як набір значень у матричному вигляду, і всі дії над цими значеннями також проходять в рамках лінійної алгебри.

Однак в платформу .NET немає вбудованого алгоритму для роботи з матрицями. Для забезпечення ефективної та швидкої роботи з матрицями було використано бібліотеку Math.NET Numerics.

Це простий та потужний інструмент, що ефективно працює з матрицями, для підключення його до проекту потрібно завантажити Nuget пакет або ж додати .dll файл бібліотеки та використати у файлі наступний код для підключення:

```
using MathNet.Numerics.LinearAlgebra;
using MathNet.Numerics.LinearAlgebra.Double;
```

5.2 Розробка тестового середовища

Під час проектування тестового середовища було виділено ряд моментів, які є особливо важливими під час розробки тестового середовища. Перший з них – структура управління камерою.

Для забезпечення управління камерою було створено окремий скрипт CameraMovement.cs, який відповідав за всі рухи камерою. Особливістю даного скрипта є адаптація рухів до пристроїв з сенсорним дисплеєм. Це дозволяє використовувати дану модель камери ефективно навіть при грі на мобільних пристроях.

Задля спрощення управління камерою в багатьох площинах, кожна з них була розділена на управління конкретним об’єктом.

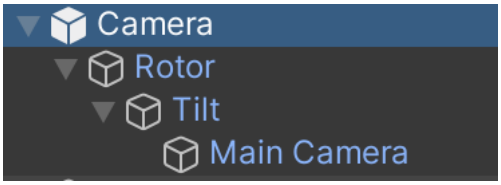

Рисунок 5.2 – структура камери в тестовому середовищі

Управління рухами камери забезпечується з базового об’єкту (виділений на рисунку). Код, написаний для управління, досить обширний і включає наступні можливості:

- Переміщення
- Приближення/Віддалення
- Поворот

Кожен з варіантів реалізовано за допомогою окремого методу, приклад реалізації методу повороту наведено нижче:

```
private void MobileCameraRotate()
   {
      if (!rotate) return;
      if (Input.touchCount > 1)
      {
         Touch touch0 = Input.GetTouch(0);
         Touch touch1 = Input.GetTouch(1);
         float deltaRotate;
         Touch leftTouch = new Touch();
         Touch rightTouch = new Touch();
         if (touch0.position.x < touch1.position.x)
         {
            leftTouch = touch0;
            rightTouch = touch1;
         }
         else if (touch0.position.x > touch1.position.x)
         {
            leftTouch = touch1;
            rightTouch = touch0;
         }

         if (leftTouch.deltaPosition.y >= 0 && rightTouch.deltaPosition.y <= 0)
         {
            deltaRotate=((Mathf.Abs(leftTouch.deltaPosition.y)                    +
Mathf.Abs(rightTouch.deltaPosition.y)) /
                         Screen.height) *
                        1080;
            rotor.transform.Rotate(0, -deltaRotate * smoot * mobileRotSpeed, 0);
         }
         else if (leftTouch.deltaPosition.y <= 0 && rightTouch.deltaPosition.y >= 0)
         {
```

```
            deltaRotate = ((Mathf.Abs(leftTouch.deltaPosition.y) +
Mathf.Abs(rightTouch.deltaPosition.y)) /
                          Screen.height) *
                         1080;
            rotor.transform.Rotate(0, deltaRotate * smoot * mobileRotSpeed, 0);
        }
    }
```

Окрім управління камерою важливу увагу в проектуванні системи було приділено створенню зручного інтерфейсу користувача.

Для його створення в Unity було використано UI елементи, вбудовані в ігровий рушій. Згідно з макетом інтерфейс було розділено на 5 умовних частин, однак з точки зору ігрового рушія було побудовано два види взаємодії – з інтерфейсом та з ігровим полем. На діаграмі використання, яку можна побачити в додатку Г, показано можливості зміни стану та впливу на ігрові об’єкти через інтерфейс користувача та ігрові об’єкти.

Як можна побачити з діаграми, багато елементів інтерфейсу тісно пересікаються з елементами, що знаходяться на ігровому полі, вибір ігрового об’єкта впливає на інтерфейс, а використання UI може вплинути на ігрове поле. Така досить складна система реалізувати зручний для користувачів функціонал та забезпечує комфортний ігровий процес, оскільки при необхідності можна використовувати різні, більш зручні, варіанти надання наказів.

Всі накази, які можна віддати у грі було сформовані у чітку структуру. Це необхідно через потребу зрівняння можливостей гравця і бота, оскільки у випадку, якщо гравець матиме більше інструментів для управління бот не зможе йому протистояти через неможливість віддати певні накази. Кожен з наказів має власний код, що дозволяє зручно синхронізувати з сервером дані і передавати їх між гравцями.

Для прийому та передачі даних, зв’язку з модулем веб-запитів та іншого функціоналу управління в рамках ігрового процесу використовується модуль контролерів.

Структурно контролери розділені на дві частини – контролер, що відповідає за управління юнітами та контролер, що відповідає за управління будівлями. Однак для синхронізації взаємодії дані контролери обмінюються інформацією.

Основний функціонал в контролерах викликається у функції Update. Це спричинено особливостями функціонування компонентів, що наслідують клас MonoBehaviour. Життєвий цикл (повний) компонентів на рисунку 5.3.

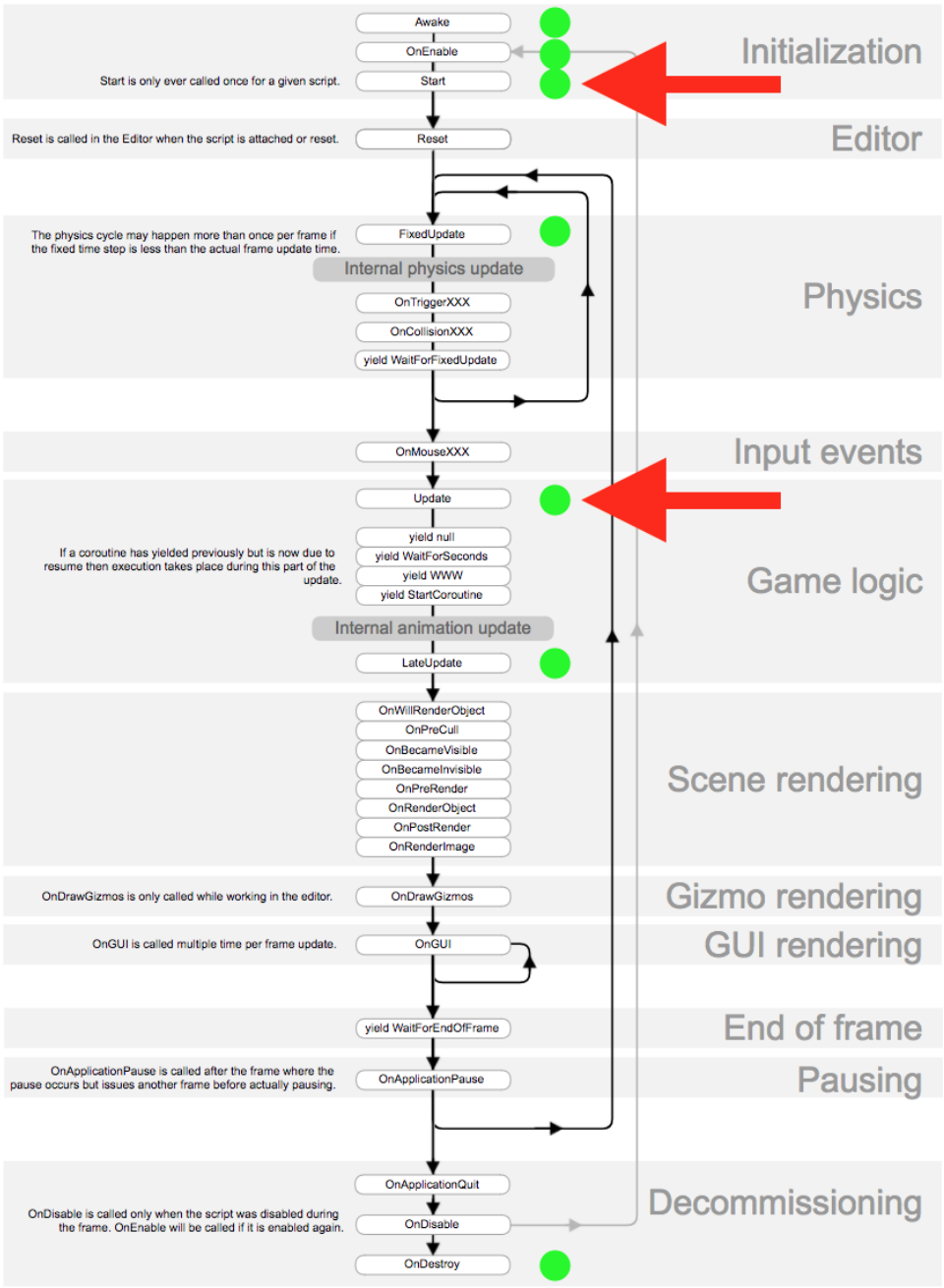

Рисунок 5.3 – Життєвий цикл в Unity

На рисунку зеленим позначено методи, доступні для використання у скриптах. Якщо додати логіку в ці методи Unity виконає її на етапі життєвого циклу, до якого відноситься конкретний метод.

Червоними стрілками позначено методи, які по замовчуванню створюються у всіх класах, які наслідують MonoBehaviour. Вигляд створеного скрипта можна побачити на рисунку 4.4.

```
1   using System;
2   using UnityEngine;
3
4   namespace RealTimeStrategy
5   {
        No asset usages
6       public class Peasant : MonoBehaviour
7       {
            Event function
8           private void Start()
9           {
10          }
11
            Event function
12          private void Update()
13          {
14          }
15      }
16  }
```

Рисунок 5.4 – Пустий клас (скрипт) в Unity

В контролерах дії, які проводить користувач, напряму впливають на ігрову логіку. Саме метод Update згідно схеми є першим методом, що активується після вводу користувачів на кожному фреймі.

Суть контролерів полягає у тому, що кожен фрейм проводиться перевірка на те, які дії були виконані гравцем і що потрібно змінити перед рендером (якщо розглянути схему 5.3 можна побачити що рендер проводиться після попередніх етапів). Тому метод Update – самий навантажений логікою об’єкт класу, в якому практично всі методи використовуються.

```
private void Update()
{
    if (!EventSystem.current.IsPointerOverGameObject() && Input.GetMouseButtonUp(0) && Input.touchCount == 0)
    {
        var isDoubleClick = Time.time - _lastClickTime < CatchTime;
        Debug.Log( message: "Catch Double Click: " + isDoubleClick);
        var ray = camera.ScreenPointToRay(Input.mousePosition);
        if (!MultiSelect && Physics.Raycast(ray, out var hit,  maxDistance: Mathf.Infinity, unitLayerMask))
        {
            HandleUnitClick(hit, isDoubleClick);
        }
        else if (!MultiSelect && Physics.Raycast(ray, out var hitGround,  maxDistance: Mathf.Infinity, terrainLayerMask))
        {
            HandleMoveToPoint(hitGround);
        }
        else if (MultiSelect && Physics.Raycast(ray, out var selectGround :RaycastHit ,  maxDistance: Mathf.Infinity, terrainLayerMask))
        {
            HandleMultiSelection(selectGround);
        }
        _lastClickTime = Time.time;
    }
}
```

Рисунок 5.5 – Метод Update в UnitController

Спочатку в методі перевіряється, чи натискання не було над елементом інтерфейсу (таким чином забезпечуються управління виключно UI елементами незважаючи на те, що знаходиться «під ними» на ігровому полі»).

Після проходження перевірки за допомогою метода ScreenPointToRay з об’єкта камери створюється луч у напрямку, в якому повернена камера. Далі іде ряд перевірок на пересікання певних ігрових елементів – бойових одиниць або ігрового поля.

Якщо обрана бойова одиниця – викликається наступний метод для обробки натискання на неї (рис. 5.6). У інших випадках також викликаються специфічні обробники.

Окрім цього в даній частині коду можна побачити використання логів в консоль Unity. За допомогою статичного класу Debug можна викликати статичний метод Log, що дозволить вивести в консоль потрібне повідомлення.

Окрім Debug варто відмітити використання класу Time. Даний клас відповідає за показники поточного часу в екосистемі Unity.

```
private void HandleUnitClick(RaycastHit hit, bool isDoubleClick)
{
    var unit = hit.transform.gameObject.GetComponent<IUnit>();
    if (unit != null)
    {
        var cUnit = _allUnits.Find( match: x :ControlledUnit => unit.Id == x.UnitId);
        var alreadySelected :bool = _controlledUnits // List<ControlledUnit>
            .FirstOrDefault( predicate: x :ControlledUnit => x.UnitId == cUnit.UnitId) != null;
        if (!_controlledUnits.Any())
        {
            AddUnitToSelection(cUnit);
        }
        else if (alreadySelected)
        {
            UnselectAll();
            Debug.Log( message: ("cUnit.LastSelectedTime; " + cUnit.LastSelectedTime));
            if (isDoubleClick && Time.time - cUnit.LastSelectedTime <= CatchTime)
            {
                AddUnitsToSelection(_allUnits.Where(x :ControlledUnit =>
                    x.Unit.GetTypeName() == unit.GetTypeName() &&
                    Vector3.Distance( a: x.Object.transform.position, b: cUnit.Object.transform.position) < 20));
            }
            else
            {
                AddUnitToSelection(cUnit);
            }
        }
        else
        {
            UnselectAll();
            AddUnitToSelection(cUnit);
        }
    }
}
```

Рисунок 5.6 – Обробка вибору бойових одиниць

Для синхронізації з сервером було створено два окремі модулі, пріоритетна синхронізація під час ігрового процесу проходить в класі SignalRConnector.

Загалом схема роботи в Unity з бібліотекою така сама як і веб-застосунках, оскільки повна сумісність дозволяє повноцінно використовувати клієнт для взаємодії з сервером. На клієнті обробка повідомлень від сервера проводиться за допомогою патерну Підписник, коли всередині скрипта методи підписуються на події отримання даних від серверу.

5.3 Розробка серверної частини

При розробці робочого прототипу проекту для кожного мікросервісу, окрім мікросервісу аналізу та штучного інтелекту, було створено схему бази даних. Приклад схеми бази даних для мікросервісу Game Data можна переглянути у додатку З.

Структура бази даних обширна, оскільки необхідно описати весь ігровий процес. Центральною таблицею у структурі є GameHistory, яка зберігає дані кожного успішного або помилкового запиту на створення нової гри. До цієї таблиця мають пряме відношення майже всі таблиці в рамках мікросервісної бази даних. Помилки під час гри записуються в окрему таблицю GameProcessIssues. Сторона, що приймали участь в грі (неважлива їх взаємодія) зберігаєтсья в таблиці ConflictSides. Якщо грав гравець, то в таблиці буде заповнено поле PalyerID.

Ігрові дані щодо обєктів та їх характеристик зберігаються в таблицях WarriorLibrary та BuildingLibrary. Також класифіковано всі ігрові події та накази, всі дані формалізовано у вигляду таблиць ActionType та EventType.

Враховуючи особливості функціонування мікросервісної архітектури, деякі поля, наприклад PalyerID, що на перший погляд мали би бути зовнішніми ключами, не посилаються на таблиці, оскільки дані збережені в іншому сервісі. Зв’язування даних відбувається в компоненті превинної обробки даних (API Gateway), а синхронізація – через брокер повідомлень (Service bus).

Для розробки бази даних було використано підхід Code-first [30]. Цей підхід заключається в тому, що для створення таблиць і загалом бази даних використовуються моделі, які написані на мові C#. Перевагою цього підходу є використання для розробки тільки C#. Моделі для отримання даних в будь-якому випадку потрібно буде створювати, але при використанні іншого підходу доведеться робити одну і ту саму роботу двічі: створення моделей та створення таблиць в базі даних за допомогою.

Хоча цей підхід має і певні недоліки: при такому підході швидкість роботи з базою даних дещо менша, оскільки запити до сховища даних виконуються з використання LINQ to Database. LINQ (Language-Integrated Query) – це частина мови програмування C#, яка додає можливості повнофункціонального та зручного підходу в створенні запитів до будь яких колекції даних. Також LINQ допомагає полегшити обробку даних та обробляти дані. В певній мірі це схоже на розширену версію SQL, інтегровану в C#.

Запити, написані на LINQ to Database для отримання даних з бази транслюються в SQL. Цей процес залежить від драйверу, який використовується в фреймворку, що забезпечує доступ до даних. Драйвер не підтримує всі сценарії, що робить SQL в певних випадках більш функціональним. До того ж, часто при написанні запитів на LINQ драйвер транслює його не найоптимальнішим шляхом, що зменшує швидкодію у порівнянні з скриптами, написаними вручну.

Для використання LINQ потрібно перш за все зв’язати дані моделі з таблицями в базі даних. Не дивлячись на те, що таблиці створюються з моделей, для їх підключення потрібно додати їх до файлу налаштувань. Для створення будь яких налаштувань потрібно підключити базу даних до проекту. Додавання будь яких сервісів та модулів виконується в Startup файлі, що використовується для створення веб-хосту в файлі Program. Така структура є стандартно для проектів, розроблених з використанням .NET Core версії 2 і вище. В Startup файлі виконується підключення всіх middleware, websocket та обробників запитів. При небхідності додається авторизація та аутентифікація користувача. Якщо веб-застосунок потребує використання бази даних або стороннього модуля, він має бути доданим до колекції сервісів, що ініціалізується перед початковим запуском хосту.

Для підключення бази даних до Startup клас потрібно використати метод AddDbContext. Цей метод застосовується для IServiceCollection і додає сховище даних. AddDbContext є generic методом, де типом, що передається в

метод, є клас, який наслідує базовий клас для контексту бази даних DbContext.

В даному файлі ініціалізуються всі налаштування для підключення бази даних. Кожна з таблиць підключається як DbSet. Приклад підключення таблиці Buildings:

public DbSet<Building> Buildings{ get; set; }

Дана властивість класу є generic-властивістю, в яку передається клас моделі. По стандартним налаштуванням Entity Framework всі властивості, що є класі автомапінгом додаються до SQL запиту, що створює таблицю. При необхідності особливих налаштувань для мапінгу поля використовується анотація.

Деякі можливості анотації при створенні класів моделей [20]:

- ключове поле: [Key];
- поле з забороненим null-значеням: [Required];
- поле, яке не буде додано в модель: [NotMapped];
- поле, що позначає зовнішній ключ для об’єкту, і використовується для зв’язування даних: [ForeignKey("BuildingId")];
- поле для синхронізації даних: [Timestamp];
- уточнення особливостей створення поля в таблиці бази даних: [Column(TypeName = "decimal(20, 10)")]. В даному випадку кількість знаків при створенні колонки в таблиці бази даних буде обмежена.

5.4 Результати тестування

Проблемним було налаштування навчання системи, адже на відміну від типових задач, для яких використовують нейронні мережі, однозначно «правильного» варіанту не можна надати системі. Однак, система функцій, яка надає можливість отримувати дані про поточний стан військ, дозволяє

моделювати ситуації з різними варіантами і надавати системі «правильний» варіант з розрахунку максимальної кількості балів. Це ускладнює навчання моделі з точки зору навантаження, однак дозволяє робити правильні висновки. Також, додано можливість коригувати дії за допомогою вказування правильного результату.

Основною метрикою, з допомогою якої вимірювалась ефективність системи, була оцінка ефективності використання бойових одиниць. Під час зіткнення бойових одиниць розраховувалась загальна оцінка групи військ, що вступає в територію контакту. Після зміни території зіткнення – переходу фокусу на іншу точку карти або знищення всіх бойових одиниць однієї з сторін – підраховується загальна оцінка. Загалом ця оцінка є різницею втрат гравців в поточній зоні, якщо втрати менші – дії в зоні були успішними. Ефективне використання військ в разі використання таких метрик можливо врахувати навіть при повній втраті бойових одиниць в межах території, оскільки вони можуть нанести пошкоджень ворожим бойовим одиницям більше, ніж мають. Ефективність дій також можна визначити величиною оцінки – чим вона вища – тим вища різниця між втратами гравця та супротивника, а значить одиниці було використано максимально ефективно.

Для тестування використано декілька типових, попередньо заданих алгоритмів, що часто використовуються в стратегіях. Це види ботів з умовними позначенням «атакуючий», «захисний» та «збалансований».

Далі в табл. 1 наводиться порівняльна характеристика результатів поєдинків бота під керуванням нейронної мережі та одного з ботів, які мають визначений алгоритм.

Для визначення ефективності навчання створено 3 варіанти моделі. Перший навчався за результатами тестувальників («правильний» варіант відповіді нейромережі задавався людиною). Другий варіант навчався виключно з використанням симуляції та підрахунку оцінки. Третій є комбінацією навчальний даних, які були використані.

Таблиця 5.1 – Результати тестів

	атака	захист	баланс
1	67%	61%	64%
2	72%	65%	71%
3	75%	66%	73%

При аналізі результатів можна побачити, що ефективність нейронної мережі в порівнянні з стандартними ботам навіть на невеликих об’ємах тестових даних досить висока. Проти агресивного агенту ефективність вища, оскільки в бою набагато простіше отримати перевагу – бот не чекає підкріплень і втрачає одиниці невеликими групами. Нижчі результати з захисним ботом пов’язані з частим відступом останнього за межі території зіткнення, через необхідність отримання більшої маси військ або відсутності стратегічної точки для оборони. Відступ з зони не змінює цінності бойових одиниць, тому перемоги немає.

Загалом для оцінки проведено по 10 матчів. В кожному з матчів кількість боїв була різна. Цікавими результатами також є залежність кількості боїв в матчі від навченості нейронної мережі (рис. 6). Як можна побачити з рис. 6, на початку нейронна мережа заповнена випадковими коефіцієнтами, результат відповідний – швидка поразка через помилки управління. Зі збільшенням навченості нейронної мережі кількість зіткнень – активацій тактичного штучного інтелекту – за один матч зростала, оскільки до моменту закінчення бою проходило більше часу через вищу ефективність Друга частина графіку показує ту частину навчання мережі, коли штучний інтелект став потужнішим за вбудований інтелект. Кількість зіткнень за матч почала падати, оскільки через високу ефективність використання зіткнення вели до великих втрат бойових одиниць, на відновлення яких потрібен час.

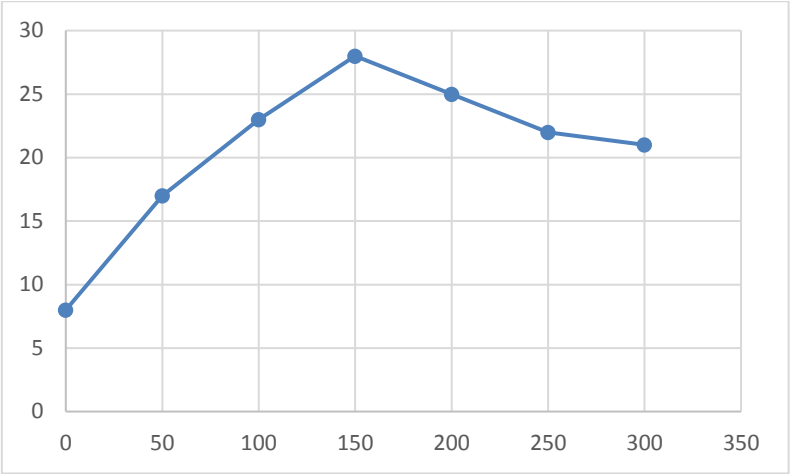

Рисунок 5.7 – Залежність кількості зіткнень від навчання нейронної мережі

Аналізуючи табл. 1 можна дійти висновку, що комбінація навчання за результатами симуляції та навчання за прикладами створеними людиною, є оптимальним варіантом. Проблема навчання тільки за симуляцією в тому, що приклади від людини мають дещо більш узагальнене бачення картини і прораховування кроків на довгий час.

Ефективність штучного інтелекту можна збільшити навчанням мережі на більшій кількості даних, а також покращити алгоритм симуляції для повної автоматизації процесу.

5.5 Висновки до розділу

Розроблено тестове середовище, яке є спрощеною стратегією в реальному часі з обмеженою кількістю типів військ та дещо спрощеними ігровими правилами.

Для роботи в тестовому середовищі створено штучний інтелект для прийняття тактичних та стратегічних рішень в управлінні. В моделі роботи штучного інтелекту активно використовувались підходи, аналогічні тим, що використовуються в згорткових нейронних мережах.

За результатами тестування можна визначити, що штучний інтелект досить ефективний проти ботів та є гнучким супротивником. Для подальшого покращення результатів потрібно збільшити обсяги тестових даних та урізноманітнити їх.

ВИСНОВКИ

У результаті виконання роботи було досліджено актуальність обраної теми. Розглянуто та проаналізовано існуючі рішення та підходи до розробки штучного інтелекту для стратегій в реальному часі, виділено основні компоненти в стратегічному ШІ та досліджено особливості розробки макро- та мікро-стратегічного інтелекту, а також технології прогнозування дій супротивника.

Опрацьовано велику кількість інформації від передових видань у сфері розробок штучного інтелекту, надано загальну інформацію щодо розробки штучного інтелекту для ігрових продуктів та наукових проектів, пов’язаних з даним жанром ігор.

На основі проаналізованої інформації обрано оптимальний варіант комбінації агентів для створення наближеного до людини та потужного штучного інтелекту. Спроектовано реалізації обраних методів, дано розгорнуте порівняння та опис переваг саме такого підходу для створення інтелекту.

Обрано платформу та технології для створення тестового середовища, необхідного для перевірки штучного інтелекту на реальному продукті. Розроблено розподілену систему для розгортання застосунку та комунікації з сервером. Розроблено структуру серверної частини, обґрунтовано використання мікросервісного підходу для забезпечення стійкості системи.

Проаналізовано сучасні методи для створення та роботи з нейронними мережами, обґрунтовано використання нейронної мережі для додатку. Розглянуто можливості для серверного навчання моделі з подальшим використанням у користувацькому додатку результатів навчання моделі.

Розглянуто переваги побудови штучного інтелекту для стратегій з використанням ієрархічної мережі постановки задач. Створено ієрархічну мережу для гри в рамках тестового середовища, проаналізовано результати роботи мережі.

Розроблено зручний для користувачів мобільних пристроїв інтерфейс та протестовано можливості його використання. Окрім інтерфейсу для мобільних пристроїв також адаптовано певний набір функціональних можливостей, які є стандартом для ігор обраного жанру з основною платформою – ПК.

ПЕРЕЛІК ПОСИЛАНЬ

1. Morris R. Workshop Summary Kasparov vs. Big Blue: The Significance for Artificial Intelligence. *ICGA Journa*l. 1997. Vol. 20.
2. Buro M. Call for AI Research in RTS Games. *Proceedings of the AAAI Workshop on AI in Games*. AAAI Press, 2004.
3. Shantia A., Begue E., Wiering, M Connectionist Reinforcement Learning for Intelligent Unit Micro Management in StarCraft. *2011 International Joint Conference on Neural Networks (IJCNN)*. San Jose, CA USA, 31 July–5 August, 2011.
4. Amado, L. Reinforcement learning applied to RTS games. 2017.
5. Niel R., Krebbers J., Drugan M., Wiering M. Hierarchical Reinforcement Learning for Real-Time Strategy Games. *ICAART 2018 - 10th International Conference on Agents and Artificial Intelligenc*. 2018.
6. Szczepanski T., Aamodt, A. Case-Based Reasoning for Improved Micromanagement in Real-Time Strategy Games. *Case-Based Reasoning for Computer Games* at the 8th International Conference on Case-Based Reasoning. Seattle, WA, USA, 20–23 July, 2009.
7. Molineaux M., Aha D., Moore P. Learning Continuous Action Models in a Real-Time Strategy Environment. *Twenty-First International Florida Artificial Intelligence Research Society (FLAIRS) Conference*, pp. 257–262. Palo Alto, CA: AAAI Press, 2008.
8. Sailer F., Buro M., Lanctot M. Adversarial Planning Through Strategy Simulation. *IEEE Conference on Computational Intelligence and Games*, pp 80–87. Piscataway, NJ: Institute for Electrical and Electronics Engineers, 2007.
9. Churchill D., Saffidine A., Buro M. Fast Heuristic Search for RTS Game Combat Scenarios. *Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 112–117. Palo Alto, CA: AAAI Press, 2012.
10. Synnaeve G., Bessière P. A Bayesian Model for RTS Units Control Applied to StarCraft. *2011 IEEE Conference on Computational Intelligence and Games*, 190–196. Piscataway, NJ: Institute for Electrical and Electronics Engineers, 2011b.
11. Gabriel I., Negru V., Zaharie D. Neuroevolution Based MultiAgent System for Micromanagement in Real-Time Strategy Games. *Fifth Balkan Conference in Informatics*, 32–39. New York: Association for Computing Machinery, 2012..
12. Stanescu M., Barriga N., Hess A., Burom M. Evaluating Real-Time Strategy Game States Using Convolutional Neural Networks, 2016.
13. Jie H., Weilong Y. A multi-size convolution neural network for RTS games winner prediction. *MATEC Web of Conferences*, 2018.
14. Aha D., Molineaux M., Ponsen M. Learning to Win: Case-Based Plan Selection in a Real-Time Strategy Game. *Case-Based Reasoning: Research and Development*, 2005.

15. Cadena P., Garrido L. Fuzzy Case-Based Reasoning for Managing Strategic and Tactical Reasoning in Star-Craft. *Advances in Artificial Intelligence,* 2011.
16. Muñoz-Avila H., Aha D. 2004. On the Role of Explanation for Hierarchical Case-Based Planning in Real-Time Strategy Games. *Advances in Case-Based Reasoning, 7th European Conference, ECCBR 2004*. Lecture Notes in Computer Science. Berlin: Springer, 2004.
17. Laagland, J. A HTN Planner for a Real-Time Strategy Game, 2008.
18. Weber B., Ontañón S. Using Automated Replay Annotation for Case-Based Planning in Games. *Case-Based Reasoning for Computer Games at the 8th International Conference on Case-Based Reasoning*, Seattle, WA, USA, 20–23 July, 2010.
19. Molineaux M., Klenk M., Aha D. Goal-Driven Autonomy in a Navy Strategy Simulation. *Twenty-Fourth AAAI Conference on Artificial Intelligence*. Palo Alto, CA: AAAI Press, 2010.
20. Weber B., Mateas M., Jhala, A. Applying Goal- Driven Autonomy to StarCraft. *Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 101–106. Palo Alto, CA: AAAI Press, 2010.
21. Kabanza F., Bellefeuille P., Bisson F., Benaskeur A., Irandoust H. Opponent Behaviour Recognition for Real-Time Strategy Games. *Plan, Activity, and Intent Recognition*: Papers from the AAAI Workshop. Technical Report WS-10-15. Palo Alto, CA: AAAI Press, 2010.
22. Weber B., Mateas M. A Data Mining Approach to Strategy Prediction. *2009 IEEE Symposium on Computational Intelligence and Games*, 140–147. Piscataway, NJ: Institute for Electrical and Electronics Engi- neers, 2009.
23. Dereszynski E., Hostetler J., Fern A., Dietterich T., Hoang, T., Udarbe, M. Learning Probabilistic Behavior Models in Real-Time Strategy Games. *Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 20–25. Palo Alto, CA: AAAI Press, 2011.
24. Hostetler J., Dereszynski E., Dietterich T., Fern A. Inferring Strategies from Limited Reconnaissance in Real-Time Strategy Games. *Conference on Unceertainty in Artificial Intelligence*, Avalon, Catalina Island, CA, 15–17 August, 2012.
25. Zhu J., Villareale J., Javvaji N., Risi S., Löwe M., Weigelt R., Harteveld C. Player-AI Interaction: What Neural Network Games Reveal About AI as Play. 2021.
26. Нейронні мережі - шлях до глибинного навчання. URL: https://codeguida.com/post/739 (дата звернення: 10.05.2021).
27. Simple Hierarchical Ordered Planner. URL: http://www.cs.umd.edu/projects/shop/index.html (дата звернення: 10.05.2021).
28. Unity vs. Unreal – Choosing a Game Engine. URL: https://gamedevacademy.org/unity-vs-unreal/ (дата звернення: 10.05.2021).
29. 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. URL:

https://www.gamasutra.com/view/feature/131503/1500_archers_on_a_288_network_.php (дата звернення: 10.05.2021).
30. Свириденко О.А. Веб-застосунок для пошуку репетитора на базі мікросервісної архітектури. URL: https://ela.kpi.ua/handle/123456789/30233 (дата звернення: 10.05.2021).
31. Designing And Implementing A Neural Network Library. URL: https://www.codeproject.com/Articles/14342/Designing-And-Implementing-A-Neural-Network-Librar#_articleTop (дата звернення: 10.05.2021).

ДОДАТОК А

Перелік публікацій

УДК 004.8 Інформаційні технології

ПЕРСПЕКТИВИ РОЗВИТКУ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ ІГОР В ЖАНРІ СТРАТЕГІЙ В РЕАЛЬНОМУ ЧАСІ

Дорогий Я.Ю.,
Кандидат технічних наук, доцент,
НТУУ «Київський політехнічний інститут» ім. Ігоря Сікорського
м. Київ, Україна

Свириденко О.А.,
Магістрант,
НТУУ «Київський політехнічний інститут» ім. Ігоря Сікорського
м. Київ, Україна

Анотація: В роботі наведено *детальний аналіз підходів та перспектив розвитку штучного інтелекту в відеоіграх. В рамках аналізу визначено основні напрямки для використання методів штучного інтелекту в стратегія. Окремо розглянуто навчання штучного інтелекту та адаптація до нових стратегій або ситуацій. Досліджено причини недостатньо активного розвитку наукових досліджень у сфері штучного інтелекту для ігор загалом та жанру стратегій в реальному часі зокрема*.

Ключові слова: штучний інтелект; стратегії в реальному часі; відеоігри; нейронні мережі.

Штучний інтелект активно використовується в різних сферах, від медицини до економіки. Однак, не дивлячись на таке активне поширення, найперспективнішою сферою для розвитку та аналізу результативності штучного інтелекту залишаються комп’ютерні ігри. Особливо серед різних жанрів варто

виокремити стратегії в реальному часі. Причиною цього є складність задач, які ставляться перед гравцем та штучним інтелектом в даному жанрі гри [1].

За глобальне управління, яку дослідники також називають макроконтроль, у бота відповідає **модуль прийняття стратегічних рішень**. За управління військами під час сутичок, або мікроконтроль, відповідає **модуль прийняття тактичних рішень**. З самого початку гравцю доступна тільки невелика частина карти – більшість потрібно розвідати, також невідома поточна ситуація з будівлями та військами супротивника. Для цього потрібна розвідка та аналіз, за які відповідає **модуль розвідки та визначення планів**. Окремо варто зазначити, що окрім вищезгаданих трьох модулів штучного інтелекту в стратегіях в реальному часі, ще існує навчання агента для коректного реагування на ситуації.

Для створення тактичного штучного інтелекту у бота дослідники та інженери використовують різні підходи, найчастішими варіантами серед яких є:

- *Навчання з підкріпленням;*
- *Пошук по ігровому дереву;*
- *Баєсова модель;*
- *Рішення на основі прецедентів;*
- *Нейронні мережі.*

Стратегічні рішення, або макроменеджмент, є високорівневою частиною штучного інтелект в стратегічних іграх і впливають на гру в довгостроковій перспективі. Наприклад, досліджена технологія або побудована будівля на початку гри може змінити ситуацію і вплинути на розвиток подій в самому кінці гри. Для визначення стратегічних дій використовують планувальні системи. Вони показали свою ефективність в реальних продуктах і наукових дослідженнях. Складність проблеми полягає у залежності стратегічних рішень від рішень супротивника. Однак використання так званої технології «туман війни» не дозволяє знати, що є на базі у супротивника та який розмір армії. Через такі проблеми рішення доводиться приймати керуючись неповними або навіть відсутніми даними, що може суттєво вплинути на наслідки. Основними підходами для розв’язання даного типу завдань є *планування на основі прецедентів,*

ієрархічне планування та *автономне досягнення цілей*. Рідше використовуються *дерева поведінки* або *еволюційні системи*.

З стратегічного планування, як окремий модуль, можна виокремити техніку визначення плану супротивника та його стратегії. Через технологію «туману війни», яку часто використовують в стратегіях, визначення планів майже завжди базується на неповній інформації, тому часто техніки використовують уже готову базу знань або навчальний модуль. Найчастіше використовують наступні підходи:

- *Дедуктивні;*
- *Абдуктивні;*
- *Ймовірнісні;*
- *Прецедентні.*

Одна з важливих проблем для штучного інтелекту в ігрових системах загалом, і в жанрі стратегій в реальному часі зокрема, є різні цілі для бізнесу та для науковців. Для бізнесу основна ціль штучного інтелекту – максимально збільшити задоволеність гравців продуктом. Для науковців основною ціллю є максимізація ефективності, отже максимальне наближення кількості перемог штучного інтелекту над гравцями до 100%. Це і є основною проблемою та причиною використання зовсім різних алгоритмів, адже гравцям в більшості випадків не хочеться програвати ботам практично всі поєдинки.

Ще одним важливим фактором є об’ємність досліджень, а отже, і їх висока ціна, оскільки стратегії в реальному часі – це реальні конфлікти, які імітуються у середовищі з спрощеними правилами, однак все ще дуже близькими до реальних. Деякі ігрові компанії все ж намагаються створювати якісний штучний інтелект і роблять це достатньо успішно. Для гри Бліцкриг 3 в 2017 році було створено тактичний штучний інтелект на основі нейронної мережі[2], який після навчання керував військами на рівні професійних гравців.

Отже, проаналізувавши окремо напрямки розвитку штучного інтелекту в ігрових стратегіях можна з впевненістю сказати, що сфера буде розвиватись через можливість в іграх даного жанру моделювати складні процеси і вивчати взаємодію між гравцями та ботами у складних ситуаціях. Серед напрямів розвитку можна

виокремити наступні: розвиток штучного інтелекту з «людською» поведінкою, покращення навичок аналізу ситуації та визначення стратегії супротивника, а також можливість активної взаємодії з гравцями-союзниками.

Література:

1. Michael Buro. (2004). Call for AI Research in RTS Games. Proceedings of the AAAI Workshop on AI in Games. AAAI Press.
2. Jichen Zhu, Jennifer Villareale, Nithesh Javvaji, Sebastian Risi, Mathias L√∂we, Rush Weigelt, Casper Harteveld. (2021). Player-AI Interaction: What Neural Network Games Reveal About AI as Play.

International Internet Conference

CERTIFICATE OF PARTICIPATION

is awarded to

Svyrydenko Oleksandr

for being an active participant in LXVI International Scientific and Practical Conference

INNOVATIONS OF THE SCIENCE XX CENTURY

17 May 2021, Ukraine, Dnipro

el-conf.com.ua

УДК 004.8 Інформаційні технології

ТАКТИЧНИЙ ШТУЧНИЙ ІНТЕЛЕКТ З ВИКОРИСТАННЯМ НЕЙРОННОЇ МЕРЕЖІ ДЛЯ СТРАТЕГІЇ В РЕАЛЬНОМУ ЧАСІ

Дорогий Я.Ю.,
Кандидат технічних наук, доцент,
НТУУ «Київський політехнічний інститут» ім. Ігоря Сікорського
м. Київ, Україна

Свириденко О.А.,
Магістрант,
НТУУ «Київський політехнічний інститут» ім. Ігоря Сікорського
м. Київ, Україна

Анотація: містить розгляд та аналіз реалізації штучного інтелекту для гри в жанрі стратегії в реальному часу з використанням нейронної мережі. Штучний інтелект не є повноцінним агентом для стратегічного, тактичного та розвідувано-аналітичного аспектів гри. Реалізація концентрується на вирішенні питань управління невеликими групами юнітів в ситуації контакту з супротивником та синхронізація дій груп з ціллю максимізації ефективності кожної окремої бойової одиниці.

Ключові слова: штучний інтелект; стратегії в реальному часі; відеоігри; нейронні мережі.

В стратегіях в реальному часі проблема побудови якісного штучного інтелекту для управління військами та загальним напрямом розвитку є проблемою для науковців ще з моменту зародження жанру. Для комерційних проектів використання сучасних підходів для створення штучного інтелекту – занадто

дорогий варіант, тому найчастіше використовується написаний статичний алгоритм, для ігрової індустрії використовують термін «заскриптований». Все це викликає негатив у користувачів ігрових продуктів, який можна уникнути використовуючи більш просунуті технології та алгоритми для створення штучного інтелекту.

Штучний інтелект в іграх жанру стратегії в реальному часі розділяють на декілька частин, які разом утворюють комплексну систему прийняття рішенн. Для розробки тактичного штучного інтелекту дослідники та комерційні компанії використовують різні алгоритми, серед яких розглядалися наступні варіанти:

- Пошук по ігровому дереву [1];
- Баєсова модель [2];
- Рішення на основі прецедентів [3];
- Нейронні мережі [4].

В ході аналізу виявлено, що нейронні мережі є одним з найбільш перспективних варіантів для штучного інтелекту в стратегіях, оскільки є найбільш гнучкими та адаптованими до непередбачуваних ситуацій.

Створюваний агент буде використовувати доступні дані з певної зони – по суті те, що бачив би гравець. Отримані дані є набором координат і характеристик військ, які потрібно розбити на визначені невеликі частини. Управління – віддавання наказів – буде реалізовано для груп бойових одиниць, де кожен квадрат – окрема група (при відсутності військ накази не будуть віддаватись). Після розділу на групи умовна зона аналізу буде мати 25 під-зон (рисунок 1), в кожній з яких буде окрема група під управлінням бота. Кінцева точка наказу для пересування групи буде в рамках «зони впливу» групи, яка складає 9 кліток (позицію і сусідів).

Для прийняття рішення щодо подальших дій групи для кожної окремої зони буде проводитись 2 окремі дослідження – для аналізу зон впливу та аналізу ситуації в безпосередньому місці розміщення групи.

Фактичним результатами, з точки зору людини, будуть дані про можливість утримання поточної позиції виключно силами, які в ній знаходяться. Оскільки для

аналізу потрібно, щоб у кожній зоні були сусіди з усіх сторін, використано аналог техніки Padding, яка часто використовується для аналізу зображень в згорткових нейронних мережах [5].

Після аналізу дані про оточуючі зони та поточний стан у конкретній зоні передаються в нейронну мережу, яка в свою чергу надає на вихід дані про напрям руху (або утримання поточної зон) і режим руху. Як приклад, результатом для зони 3 є напрям, що можна виразити вектором [1;-1] і режимом переходу 0. Відступ необхідний через велику концентрацію одиниць супротивника, хоча в зоні перевага була за військами агента.

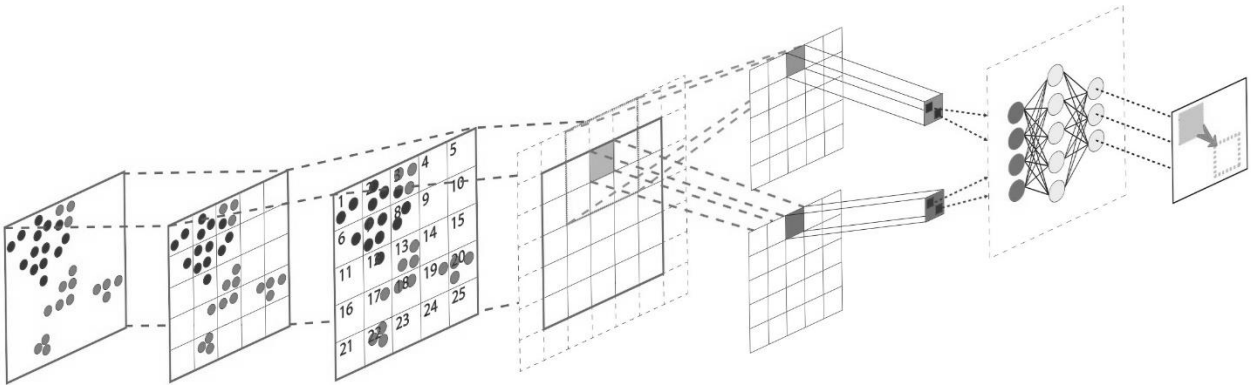

Рисунок 1 – Процес аналізу для зони 3

Література:

1. Synnaeve, G., and Bessière, P. 2011b. A Bayesian Model for RTS Units Control Applied to StarCraft. In Proceedings of the 2011 IEEE Conference on Computational Intelligence and Games.

2. Sailer, F.; Buro, M.; and Lanctot, M. 2007. Adversarial Plan- ning Through Strategy Simulation. In Proceedings of the IEEE Conference on Computational Intelligence and Games, 80–87.

3. Szczepanski, T., and Aamodt, A. 2009. Case-Based Reasoning for Improved Micromanagement in Real-Time Strategy Games.

4. Stanescu, Marius & Barriga, Nicolas A. & Hess, Andy & Buro, Michael. (2016). Evaluating Real-Time Strategy Game States Using Convolutional Neural Networks.

5. Wiranata, Aldi & Wibowo, Suryo Adhi & Patmasari, Raditiana & Rahmania, Rissa & Mayasari, Ratna. (2018). Investigation of Padding Schemes for Faster R-CNN on Vehicle Detection.

ДОДАТОК Б

Вихідний код застосунку

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEditor;
using UnityEngine;
using UnityEngine.AI;

public abstract class Person : MonoBehaviour, IUnit
{
    [SerializeField] public NavMeshAgent agent;
    [SerializeField] public GameObject pointer;
    [SerializeField] public float baseMoveSpeed;
    [SerializeField] public float baseRotationSpeed = 10;
    public Guid Id { get; private set; }
    public float Health { get; set; }
    public GameObject Owner { get; set; }
    public float Speed
    {
        get => baseMoveSpeed;
        set { }
    }

    public UnitState CurrentState { get; set; }
    public abstract bool SetState(UnitState newState);
    public abstract string GetTypeName();

    protected Vector3? targetPos;
    protected Animator animator;

    private void Awake()
    {
        Id = Guid.NewGuid();
        animator = GetComponent<Animator>();
    }

    // Start is called before the first frame update
    void Start()
    {
        pointer.SetActive(false);
    }

    // Update is called once per frame
```

```
    void Update()
    {
        var distance = Vector3.Distance(targetPos.GetValueOrDefault(),
transform.position);
        if (targetPos != null && distance > 0.7f)
        {
            var position = gameObject.transform.position;
            var simpleVector = targetPos.GetValueOrDefault() - position;

            if (false && RotateToTarget(simpleVector))
            {
                SetState(UnitState.FastMove);
                var moveVector = simpleVector / distance * (Time.deltaTime * Speed);
                transform.position += moveVector;
            }

            SetState(UnitState.FastMove);
            agent.SetDestination(targetPos.Value);
        }
        else
        {
            targetPos = null;
            SetState(UnitState.Stay);
        }
    }

    public bool RotateToTarget(Vector3 simpleVector)
    {
        var targetAngle = Vector3.Angle(Vector3.forward, simpleVector.normalized);
        if (targetPos != null && targetPos.Value.x < transform.position.x)
        {
            targetAngle = 360 - targetAngle;
        }

        var delta = Mathf.DeltaAngle(transform.eulerAngles.y, targetAngle);

        if (Mathf.Abs(delta) > 3)
        {
            SetState(UnitState.Move);

            var rotation = Time.deltaTime * baseRotationSpeed;
            var isLeftMove = delta < 0;

            var eulerAngles = transform.eulerAngles;
```

```
            eulerAngles = new Vector3(eulerAngles.x, eulerAngles.y + (isLeftMove ? -
rotation : rotation),
                eulerAngles.z);
            transform.eulerAngles = eulerAngles;

            return false;
        }

        return true;
    }


    public void Select()
    {
        pointer.SetActive(true);
    }

    public void Unselect()
    {
        pointer.SetActive(false);
    }

    public void MoveToPoint(Vector3 point)
    {
        targetPos = point;
    }

    public void Attack()
    {
        throw new NotImplementedException();
    }

    public void ReceiveDamage()
    {
        throw new NotImplementedException();
    }

    public void SetMaterial()
    {
        throw new NotImplementedException();
    }

}
```

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using Common;
using Controls.Units;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.SceneManagement;
using Object = UnityEngine.Object;

public class UnitController : MonoBehaviour
{
    [SerializeField] private new Camera camera;
    [SerializeField] private LayerMask unitLayerMask;
    [SerializeField] private LayerMask terrainLayerMask;

    private Player _owner;
    private float _lastClickTime;
    private const float CatchTime = .25f;

    // private (GameObject, IUnit)? _selectedUnit = null;
    private readonly List<ControlledUnit> _controlledUnits = new
List<ControlledUnit>();
    private readonly List<ControlledUnit> _allUnits = new List<ControlledUnit>();

    #region MultiSelect

    private bool _multiSelect = false;
    public Vector3? MultiSelectPoint { get; set; }

    public bool MultiSelect
    {
        get => _multiSelect;
        set
        {
            Debug.Log($"Multi set to: {value}");
            if (value && !_multiSelect)
            {
                ClickUnselect();
            }

            _multiSelect = value;
        }
    }
```

```
    #endregion

    #region UIHandlers

    public void ClickMultiSelect()
    {
        MultiSelect = !MultiSelect;
    }

    public void ClickUnselect()
    {
        // TODO: UI need to be updated
        UnselectAll();
    }

    #endregion

    private void Awake()
    {
        _allUnits.AddRange( Utils.FindGameObjectsWithLayer(8)
            .Where(x => x.TryGetComponent(out IUnit unit))
            .Select(x => new ControlledUnit(x.GetComponent<IUnit>(), x,
x.GetComponent<IUnit>().Id))
            .ToList());
    }

    private void Start()
    {
        _owner = Player.players.FirstOrDefault(x => x.IsCurrentPlayer);
    }

    private void Update()
    {
        if (!EventSystem.current.IsPointerOverGameObject() &&
Input.GetMouseButtonUp(0) && Input.touchCount == 0)
        {
            var isDoubleClick = Time.time - _lastClickTime < CatchTime;
            Debug.Log("Catch Double Click: " + isDoubleClick);
            var ray = camera.ScreenPointToRay(Input.mousePosition);
            if (!MultiSelect && Physics.Raycast(ray, out var hit, Mathf.Infinity,
unitLayerMask))
            {
                HandleUnitClick(hit, isDoubleClick);
            }
```

```
        else if (!MultiSelect && Physics.Raycast(ray, out var hitGround,
Mathf.Infinity, terrainLayerMask))
        {
            HandleMoveToPoint(hitGround);
        }
        else if (MultiSelect && Physics.Raycast(ray, out var selectGround,
Mathf.Infinity, terrainLayerMask))
        {
            HandleMultiSelection(selectGround);
        }
        _lastClickTime = Time.time;
    }
}

private void HandleUnitClick(RaycastHit hit, bool isDoubleClick)
{
    var unit = hit.transform.gameObject.GetComponent<IUnit>();
    if (unit != null)
    {
        var cUnit = _allUnits.Find(x => unit.Id == x.UnitId);
        var alreadySelected = _controlledUnits
            .FirstOrDefault(x => x.UnitId == cUnit.UnitId) != null;
        if (!_controlledUnits.Any())
        {
            AddUnitToSelection(cUnit);
        }
        else if (alreadySelected)
        {
            UnselectAll();
            Debug.Log(("cUnit.LastSelectedTime; " + cUnit.LastSelectedTime));
            if (isDoubleClick && Time.time - cUnit.LastSelectedTime <= CatchTime)
            {
                AddUnitsToSelection(_allUnits.Where(x =>
                    x.Unit.GetTypeName() == unit.GetTypeName() &&
                    Vector3.Distance(x.Object.transform.position,
cUnit.Object.transform.position) < 20));
            }
            else
            {
                AddUnitToSelection(cUnit);
            }
        }
        else
        {
            UnselectAll();
```

```
            AddUnitToSelection(cUnit);
        }
    }
}

#region Helpers

private void AddUnitToSelection(ControlledUnit cUnit)
{
    _controlledUnits.Add(cUnit);
    cUnit.LastSelectedTime = Time.time;
    cUnit.Unit.Select();
}

private void AddUnitsToSelection(IEnumerable<ControlledUnit> cUnits)
{
    var controlledUnits = cUnits.ToList();
    _controlledUnits.AddRange(controlledUnits);
    foreach (var cUnit in controlledUnits)
    {
        cUnit.LastSelectedTime = Time.time;
        cUnit.Unit.Select();
    }
}

private void UnselectAll()
{
    _controlledUnits.ForEach(x => { x.Unit.Unselect(); });
    _controlledUnits.Clear();
}

private void UnselectById(Guid id)
{
    // TODO: probably can remove null-check and improve performance for this
method
    var unitToUnselect = _controlledUnits.FirstOrDefault(x => x.UnitId == id);
    if (unitToUnselect == null)
    {
        Debug.Log("Check UnselectById method!");
    }
    else
    {
        unitToUnselect.Unit.Unselect();
        _controlledUnits.Remove(unitToUnselect);
    }
```

```
}

private Vector3 GetMultiMovePositionModifier(int index)
{
   if (index < 5)
   {
      return MultiMoveCombinator(index);
   }

   switch (index)
   {
      case 5: return MultiMoveCombinator(1, 2);
      case 6: return MultiMoveCombinator(2, 3);
      case 7: return MultiMoveCombinator(3, 4);
      case 8: return MultiMoveCombinator(4, 1);
      case 9: return MultiMoveCombinator(1, 1);
      case 10: return MultiMoveCombinator(2, 2);
      case 11: return MultiMoveCombinator(3, 3);
      case 12: return MultiMoveCombinator(4, 4);
      case 13: return MultiMoveCombinator(1, 1, 2, 2);
      case 14: return MultiMoveCombinator(2, 2, 3, 3);
      case 15: return MultiMoveCombinator(3, 3, 4, 4);
      case 16: return MultiMoveCombinator(4, 4, 1, 1);
      case 17: return MultiMoveCombinator(1, 1, 2);
      case 18: return MultiMoveCombinator(1, 2, 2);
      case 19: return MultiMoveCombinator(2, 2, 3);
      case 20: return MultiMoveCombinator(3, 3, 2);
      case 21: return MultiMoveCombinator(3, 3, 4);
      case 22: return MultiMoveCombinator(4, 4, 3);
      case 23: return MultiMoveCombinator(4, 4, 1);
      case 24: return MultiMoveCombinator(1, 1, 4);
      default: throw new NotImplementedException();
   }
}

private Vector3 MultiMoveCombinator(params int[] ar)
{
   Vector3 Directions(int i)
   {
      switch (i)
      {
         case 0: return Vector3.zero;
         case 1: return Vector3.left;
         case 2: return Vector3.forward;
         case 3: return Vector3.right;
```

```
            case 4: return Vector3.back;
            default: throw new NotImplementedException();
        }
    }
    return ar.Aggregate(Vector3.zero, (a1, a2) => a1 + Directions(a2));
}

#endregion

private void HandleMultiSelection(RaycastHit selectGround)
{
    if (MultiSelectPoint is null)
    {
        MultiSelectPoint = selectGround.point;
    }
    else
    {
        // select all units
        var xs = new[] {MultiSelectPoint.Value.x, selectGround.point.x}.OrderBy(x =>
x).ToArray();
        var zs = new[] {MultiSelectPoint.Value.z, selectGround.point.z}.OrderBy(z =>
z).ToArray();

        AddUnitsToSelection(_allUnits
            .Where(unit =>
            {
                var position = unit.Object.transform.position;
                return position.x >= xs[0] && position.x <= xs[1] &&
                       position.z >= zs[0] && position.z <= zs[1];
            }));

        MultiSelectPoint = null;
        MultiSelect = false;
    }
}

private void HandleMoveToPoint(RaycastHit hitGround)
{
    var pointToMove = Vector3Int.CeilToInt(hitGround.point);
    pointToMove.y = 0;
    const int unitSize = 3;
    for (var i = 0; i < _controlledUnits.Count; i++)
    {
        _controlledUnits[i].Unit.MoveToPoint(
            pointToMove +
```

```
            GetMultiMovePositionModifier(i % 25) * unitSize +
            MultiMoveCombinator(i / 25) * (5 * unitSize));
        }
    }
}

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Building : MonoBehaviour
{
    public Vector2Int size  = Vector2Int.one;

    public GameObject state1;
    public GameObject state2;
    public GameObject state3;

    private Renderer _renderer;
    private protected bool Ready;

    private void Awake()
    {
        _renderer = GetComponentInChildren<Renderer>();
    }

    public void SetTransparent(bool available)
    {
        _renderer.material.color = available ? Color.green : Color.red;
    }

    public void OnBuildingSet()
    {
        _renderer.material.color = Color.white;
        StartCoroutine(Build());
    }

    private void OnDrawGizmos()
    {
        for (int x = 0; x < size.x; x++)
        {
            for (int y = 0; y < size.y; y++)
            {
                Gizmos.color = (x + y) % 2 == 0 ? Color.cyan : Color.gray;
```

```
            Gizmos.DrawCube(transform.position + new Vector3(x, 0, y),new
Vector3(1, .2f ,1));
          }
       }
   }

   public IEnumerator Build()
   {
      state3.SetActive(false);
      state1.SetActive(true);

      yield return new WaitForSeconds(3);
      state2.SetActive(true);
      state1.SetActive(false);

      yield return new WaitForSeconds(3);
      state3.SetActive(true);
      state2.SetActive(false);

      Ready = true;
   }
}
```